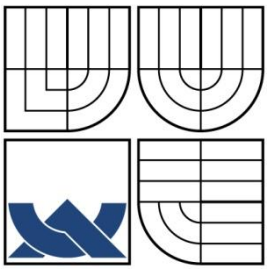
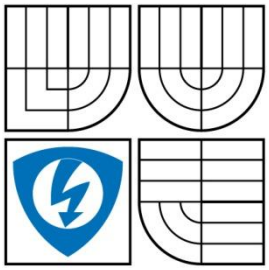


VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ
ÚSTAV MIKROELEKTRONIKY

FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION
DEPARTMENT OF MICROELECTRONICS

NÁVRH A REALIZACE MATEMATICKÝCH OPERACÍ V OBVODECH FPGA

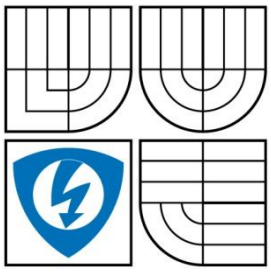
DESIGN AND REALIZATION OF MATHEMATICAL OPERATIONS IN FPGA CIRCUITS

BAKALÁŘSKÁ PRÁCE
BACHELOR´S THESIS

AUTOR PRÁCE — LUDĚK SOUKUP
AUTHOR

VEDOUCÍ PRÁCE — Ing. LUKÁŠ FUJCIK, Ph.D.
SUPERVISOR

BRNO 2009



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

Fakulta elektrotechniky a komunikačních technologií

Ústav mikroelektroniky

Bakalářská práce

bakalářský studijní obor

Mikroelektronika a technologie

Student: Luděk Soukup ***ID:*** 72888

Ročník: 3 ***Akademický rok:*** 2008/2009

NÁZEV TÉMATU:

Návrh a realizace matematických operací v obvodech FPGA

POKYNY PRO VYPRACOVÁNÍ:

Práce se bude zabývat návrhem matematických operací v obvodech FPGA. Mezi tyto matematické operace budou patřit aritmetické operace, jako je sčítání, odčítání, násobení, dělení a základní goniometrické funkce sinus, kosinus a tangens. Student bude mít za úkol prostudovat různé algoritmy pro realizaci těchto matematických operací. Vybrané algoritmy následně popsat v jazyce VHDL a implementovat do obvodu FPGA Spartan-3. V práci by mělo být uvedeno srovnání těchto algoritmů zejména z hlediska plochy a rychlosti.

DOPORUČENÁ LITERATURA:

Termín zadání: 9.2.2009 ***Termín odevzdání:*** 3.6.2009

Vedoucí práce: Ing. Lukáš Fujcik, Ph.D.

prof. Ing. Radimír Vrba, CSc.
předseda oborové rady

UPOZORNĚNÍ:

Abstrakt

Tato bakalářská práce se zabývá problematikou realizace matematických operací v prostředí digitálních obvodů architektur FPGA a ASIC. Pozornost je věnována algoritmům pro sčítání, odečítání, násobení a dělení, dále potom algoritmům pro výpočet goniometrických funkcí. Vybrané algoritmy jsou popsány v jazyce VHDL, syntetizovány a porovnány na základě získaných informací o výpočetním zpoždění a ploše, která je nutná pro jejich realizaci. V závěrečné části práce je diskutována možnost implementace těchto algoritmů.

Abstract

This bachelor's thesis deals with the issue of realization of mathematical operations in digital circuits with a focus on FPGA and ASIC architectures. The attention is focused to algorithms for addition, subtraction, multiplication and division, further to algorithms for computing trigonometric functions. Chosen algorithms are described in VHDL language, synthesized, and they are compared in terms of acquired information about computing time and area which is required for its realization. In the final part of the thesis the possibility of implementation these algorithms is discussed.

Klíčová slova

VHDL, FPGA, ASIC, Matematické operace, Goniometrické funkce, Spartan-3

Key words

VHDL, FPGA, ASIC, Mathematical operations, Trigonometric functions, Spartan-3

Bibliografická citace

SOUKUP, L. Návrh a realizace matematických operací v obvodech FPGA. Brno: Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, 2009. 57s. Vedoucí bakalářské práce Ing. Lukáš Fujcik, Ph.D.

Prohlášení autora o původnosti díla

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením vedoucího bakalářské práce, s použitím odborné literatury a dalších informačních zdrojů, které jsou všechny citovány v práci a uvedeny v seznamu použitých zdrojů. Jako autor uvedené bakalářské práce dále prohlašuji, že v souvislosti s vytvořením této bakalářské práce jsem neporušil autorská práva třetích osob, zejména jsem nezasáhl nedovoleným způsobem do cizích autorských práv osobnostních a jsem si plně vědom následků porušení ustanovení § 11 a následujících autorského zákona č. 121/2000 Sb., včetně možných trestněprávních důsledků vyplývajících z ustanovení § 152 trestního zákona č. 140/1961 Sb.

V Brně dne 3. Června 2009 ……

Luděk Soukup

Poděkování

Děkuji vedoucímu této bakalářské práce Ing. Lukáši Fujcikovi, Ph.D. za cenné odborné rady a příkladné vedení při jejím zpracování. Dále děkuji Ing. Markovi Bohrnovi za odborné konzultace při implementaci rozšiřujících modulů pro Spartan-3 Starter kit.

Obsah

1 Úvod

Oblast digitální techniky zažívá v současnosti poměrně rychlý rozvoj. Snaha vytvořit výkonnější a spolehlivější, zároveň však levnější a ekologicky šetrnější systémy, charakterizuje současné cíle v elektrotechnice. Aplikace moderních technologií představuje možnost, jak těchto cílů dosáhnout. V technické praxi se tedy stále častěji setkáváme s obvody jako DSP, FPGA nebo ASIC.

Vývoj digitálních systémů je velmi komplikovaný proces, kde nezastupitelnou roli hraje počítačová podpora návrhu - CAD systémy. Právě tyto nástroje, díky schopnosti automatizovat část procesů spojených s návrhem, urychlují vývoj digitálního systému. Ve spojitosti s těmito systémy je třeba zmínit také jazyky HDL, které umožňují efektivní popis digitálního systému a jsou zahrnuty ve většině moderních vývojových nástrojů.

Současné jazyky HDL jsou charakteristické vysokým stupněm abstrakce. Právě tato vlastnost umožňuje implementovat poměrně složité obvodové funkce bez nutnosti řešit způsob jejich realizace v cílovém obvodu. Využití těchto jazyků má ovšem také negativní důsledky. Vývojář do jisté míry ztrácí kontrolu nad vnitřní strukturou tvořeného digitálního sytému. Velmi významně se tento důsledek projevuje například u struktur realizujících matematické operace. Jejich implementace je v mnoha vývojových nástrojích možná pouhým zapsáním znaménka, reprezentujícího tuto operaci, do zdrojového kódu. Odpovídající struktura je potom automaticky vytvořena během procesu syntézy. Otázkou však zůstávají vlastnosti takto vytvořené struktury.

Tato práce si klade za cíl ověřit efektivitu matematických struktur realizovaných moderními vývojovými nástroji. Dalším cílem je, na základě známých algoritmů, vytvořit struktury umožňující alternativní implementaci matematických operací.

Problematiku realizace matematických operací v prostředí digitálních obvodů lze rozdělit do několika oblastí, jež jsou blíže popsány v jednotlivých kapitolách této práce. První kapitola představuje důvody, které vedly k realizaci tohoto projektu. Dále představuje architektury ASIC, FPGA a seznamuje s programovacím jazykem VHDL. Druhá kapitola řeší problematiku reprezentace číselných hodnot v prostředí digitálních systémů. Důraz je kladen na ty způsoby vyjádření čísel, které využívají binární kód. Třetí kapitola popisuje funkční principy algoritmů, které byly vybrány pro implementaci do cílových obvodů architektur FPGA a ASIC. Čtvrtá kapitola přechází od teoretické části k praktické. Je věnována popisu zvolených algoritmů v jazyce VHDL a jejich konkrétní realizaci. V závěru kapitoly je blíže popsán proces syntézy vytvořených zdrojových kódů. Pátá kapitola porovnává jednotlivé algoritmy na základě údajů o maximálním výpočetním zpoždění a ploše nutné k realizaci. Je zde také diskutován vliv cílové architektury na efektivitu implementovaného algoritmu. Šestá kapitola popisuje kalkulačku, vytvořenou pro ověření funkčnosti algoritmů.

1.1 Jazyk VHDL

VHDL je programovací jazyk vysoké úrovně abstrakce, který byl vyvinut pro popis a simulaci rozsáhlých digitálních systémů. Svojí povahou spadá do třídy jazyků popisujících hardware, označovaných anglickou zkratkou HDL (Hardware Description Language). Pro tento jazyk je charakteristická bohatá vyjadřovací schopnost a samozřejmě také nezávislost popisu na cílové technologii.

Popis digitálního systému jazyky HDL je velmi odlišný od klasického programování v jazycích Assembler, C nebo například Basic. Výsledný kód nepředstavuje posloupnost příkazů prováděnou procesorem, ale obvodovou strukturu – hardware. Některé elementy jazyka jsou navíc určeny pouze pro simulaci, jejich implementace do cílového obvodu se nejen nepředpokládá, ale dokonce není možná.

Historie jazyka VHDL je spojena s výzkumným projektem VHSIC (Very High Speed Integrated Circuits) ministerstva obrany Spojených států amerických, který byl zahájen v roce 1981. Vývoj jazyka probíhal v letech 1983 až 1985 ve spolupráci firem IBM, Intermetrics a Texas Instruments. Výsledkem byla základní definice jazyka nazvaného VHDL (VHSIC Hardware Description Language), který vycházel z jazyku ADA. V roce 1986 byl vývoj předán organizaci IEEE a v roce 1987 byl poprvé publikován standard jazyka VHDL pod označením IEEE Standard VHDL Language Reference Manual (IEEE Std 1076-87) známý také pod označením VHDL-87. Standard jazyka by měl být podle zvyklostí organizace IEEE každých pět let revidován, v roce 1993 byly tedy zahájeny práce na první revizi jazyka, která byla publikována v roce 1994 pod označením IEEE Std 1076-93. K dalším revizím potom došlo v letech 1999, 2000 a 2002. Tyto však nepřinesly zásadní změny a většina vývojových nástrojů tedy stále využívá standard IEEE Std 1076-93. Další podrobnosti o jazyku VHDL viz (1).

1.2 Obvody ASIC

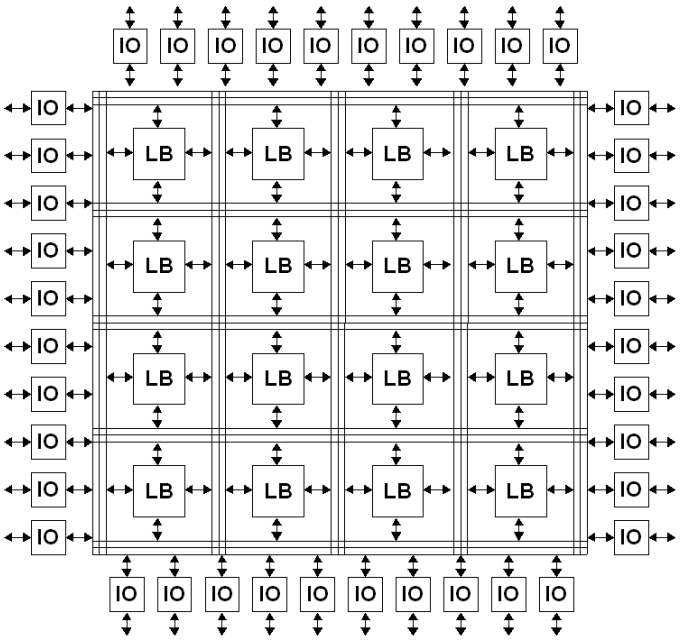
Výraz ASIC představuje akronym anglických slov „Application-Specific Integrated Circuit“. Jedná se o označení pro integrované obvody vyrobené pro specifickou aplikaci. Typickým příkladem mohou být integrované obvody v mobilních telefonech. Protiklad těchto obvodů představují standardizované obvody, které jsou využívány v mnoha různých aplikacích. Zde je možné uvést obvody řad 4000, nebo 7400.

Vývoj obvodů ASIC je velmi finančně náročný. Nemalé finanční výdaje jsou spojeny také s přípravou sériové výroby (např. litografické masky). Právě z důvodu vysoké finanční náročnosti jsou obvody ASIC využívány pouze tam, kde se předpokládá velkosériová výroba, nebo ve specifických případech (aplikace vyžadující vysokou spolehlivost).

1.3 Obvody FPGA

Obvody architektury FPGA (Field-Programmable Gate Array) patří do rodiny programovatelných logických obvodů označovaných anglickou zkratkou PLD (Programmable Logic Device). Nejčastěji jsou tyto obvody vyráběny technologií SRAM (produkty firem Altera, Xilinx a Lattice Semiconductor). Objevují se však také obvody realizované technologií Anti-Fuse (například produkty firmy Actel).

FPGA nacházejí uplatnění jednak během procesu návrhu obvodů ASIC, kdy jsou využívány k testování navržených obvodových struktur, jednak obvody ASIC nahrazují tam, kde by jejich vývoj a výroba byly nerentabilní. Typickým příkladem jsou malosériové výroby.



Obrázek 1 - zjednodušené blokové schéma vnitřní struktury obvodu FPGA

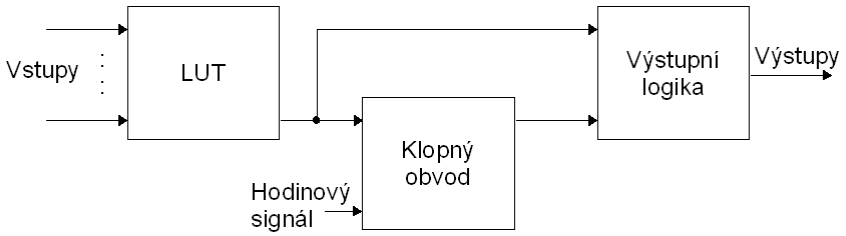
Struktura FPGA je poměrně složitá, pro vytvoření představy o funkci obvodu však postačí zjednodušené blokové schéma, které zachycuje Obrázek 1. Tato struktura obsahuje tři základní bloky:

- IO – vstupně výstupní blok, jehož úkolem je zajistit propojení vnitřní struktury obvodu s vnějším okolím. Mezi jeho funkce patří napěťové přizpůsobení signálů, proudové posílení výstupů, vzorkování vstupů atd.
- LB – logické bloky obsahují:
    - Jednu nebo více náhledových tabulek, označovaných anglickou zkratkou LUT (Look-Up Table)

- Jeden nebo více klopných obvodů
- Lokální propojovací pole
- Pomocnou logiku
- Další specifické bloky
- …

(Obrázek 2 zachycuje zjednodušenou strukturu logického bloku)

- Programovatelná horizontální a vertikální propojení - jejich funkcí je propojení různých logických bloků, nebo logického a vstupně výstupního bloku. Tyto propojení jsou v blokovém schématu znázorněny horizontálními a vodorovnými čarami.



Obrázek 2 - zjednodušená struktura logického bloku

Kromě těchto tří základních bloků, může obvod FPGA obsahovat také specializované bloky, jako jsou například násobičky, děličky, paměti, bloky pro úpravu hodinového signálu nebo mikroprocesor.

Přesná struktura a označení jednotlivých částí a bloků se může lišit v závislosti na výrobci obvodu a jeho typu. Princip a funkce jsou však ve většině případů podobné.

2 Reprezentace číselných hodnot

Algoritmus lze chápat jako princip řešení určitého problému. V rámci této práce je chápán jako způsob, kterým je možné provést výpočet určité aritmetické operace. Konkrétní realizace algoritmu a jeho efektivita je ovlivněna mnoha faktory. Mezi nejdůležitější lze zařadit cílovou technologii, popřípadě způsob vyjádření vstupních a výstupních dat.

Člověku je přirozená dekadická reprezentace dat, digitální systémy však využívají reprezentaci binární. Převod přirozeného čísla z dekadického vyjádření na binární patří mezi elementární znalosti studentů střední školy, v praxi však s přirozenými čísly nevystačíme.

2.1 Vyjádření záporných čísel

Budeme-li při výpočtech využívat množinu celých čísel, je nezbytné vyřešit způsob vyjádření záporných čísel. V následujících odstavcích jsou popsány tři nejběžněji užívané metody vyjádření.

Přímý kód

Nejpřirozenějším způsobem, jak vyjádřit záporné číslo, je přidat k absolutní hodnotě čísla znaménko mínus. Princip tohoto kódu odráží i anglický název "sign and magnitude". Tento způsob vychází z lidského chápání problematiky vyjadřování záporných čísel, a ačkoliv je pro digitální obvody poměrně nevhodný, je nezřídka využíván. Nevýhodou tohoto kódu je vznik nejednoznačnosti nuly. Ta je dána existencí kladného i záporného vyjádření.

Doplňkový kód

Někdy bývá také označován jako druhý doplněk (Second Complement). Představuje nejrozšířenější metodu vyjádření záporných čísel v binárním tvaru. Záporné číslo je získáno jako bitová negace původního čísla zvětšená o 1. Velkým přínosem je možnost využít strukturu sčítačky jak pro sčítání, tak pro odčítání, přičemž sčítání je prováděno jako přičítání záporného čísla. Dalším přínosem tohoto kódu je odstranění duplicitního vyjádření nuly.

Aditivní kód

Posledním způsobem, jak vyjádřit záporné číslo, je přičíst k jeho hodnotě známou konstantu, která bude při dalším výpočtu, ale také vyhodnocování, brána do úvahy. Výraznou nevýhodou je však fakt, že kladná čísla se liší od bezznaménkové reprezentace.

2.2 Vyjádření čísel s desetinným rozvojem

Ačkoli je množina celých čísel v některých aplikacích dostačující, v praxi bývá nejčastěji využívána množina čísel racionálních. Dva základní přístupy k vyjádření racionálních čísel představují číselné formáty pevné a plovoucí řádové čárky.

Čísla s pevnou řádovou čárkou (FX)

Hodnoty čísel zapsaných ve formátu pevné řádové čárky, anglicky označované „Fixed-Point", chápeme jako podmnožinu racionálních čísel. Jejich hodnoty lze vyjádřit vztahem:

$$X_{FX} = \frac{a}{2^b} \qquad kde\ a, b \in Z \tag{2.1}$$

Základním principem tohoto vyjádření je zachování stejného počtu binárních cifer v každém čísle. Všechna čísla tedy mají desetinnou čárku na stejné pozici. Odtud je také název. Podobně jako u jiných vyjádření, jsou nuly před první nenulovou číslicí a za poslední nenulovou číslicí nevýznamné a není důvod je uvádět.

Mezi hlavní výhody tohoto číselného systému patří možnost efektivnějšího využití systémových zdrojů. To souvisí s předem známou přesností zpracovávaných údajů. Dále možnost využít jednodušších obvodových struktur pro realizaci matematických funkcí, než v případě systému plovoucí řádové čárky. Tento způsob vyjádření je nevhodný v případě velkých rozdílů mezi zpracovávanými čísly, nebo pokud neznáme dopředu rozsah hodnot zpracovávaných dat. V těchto případech je nutné pro každé číslo alokovat vysoký počet bitů, což vede k vysoké neefektivitě.

Čísla s plovoucí řádovou čárkou (FP)

Také čísla s plovoucí řádovou čárkou, anglicky označované „Floating-Point", chápeme jako podmnožinu racionálních čísel. Na rozdíl od čísel ve formátu FX, nese každé číslo informaci o pozici desetinné čárky v sobě. Tato čísla lze vyjádřit následujícím vztahem:

$$X_{FP} = 2^e \cdot m \tag{2.2}$$

Kde *e* je hodnota exponentu a *m* je mantisa. Mantisa může být kladná i záporná, často jde o binární číslo vyjádřené v přímém kódu. Přesná specifikace tohoto formátu je do značné míry závislá na oblasti využití. Obecně je pod pojmem plovoucí řádová čárka chápán formát vycházející z normy IEEE 754.

Ve prospěch tohoto formátu hovoří jeho značná rozšířenost a to jak v programovacích jazycích, tak v hardware. Naopak mezi nevýhody patří značná složitost práce s tímto datovým formátem. Čísla v plovoucí řádové čárce nejsou ekvivalentem reálných čísel. Právě tento omyl bývá příčinou mnoha chyb.

3 Algoritmy pro implementaci matematických operací

3.1 Sčítání a odčítání

Sčítačky mají mezi aritmetickými obvody výsadní postavení. Je to dáno jejich častým využitím jak při implementaci sčítání a odčítání, tak při realizaci složitějších obvodových struktur. Jako příklad poslouží násobičky nebo děličky, kterým bude věnována pozornost v následujících podkapitolách.

Obdobně jako u většiny digitálních obvodů i zde vyvstává otázka sekvenčního či kombinačního způsobu realizace zvoleného algoritmu. Obecně platí, že při realizaci stejného algoritmu dosahují sekvenční struktury nižší plochy, kombinační struktury naopak vyšší rychlosti. Vnitřní struktura sčítaček je poměrně jednoduchá, aspekt snížení plochy obvodu tedy není natolik významný, aby vyvážil nárůst výpočetního zpoždění. Praktické využití sekvenčních struktur je tedy spíše výjimečné.

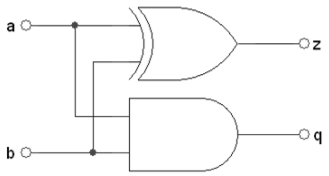
Struktura této kapitoly je volena tak, aby na příkladech jednobitových sčítaček byl vysvětlen princip sčítání v binární soustavě. Jednobitové sčítačky jsou dále využity jako základní prvek sčítačky s postupným přenosem Ripple-Carry. Následovat budou algoritmy Carry-Chain, Carry-Skip, Carry-Select, Carry-Lookahead a Brent-Kung. Ty reprezentují kvalitativně dokonalejší celky, využívající generátory rychlého přenosu. V závěru kapitoly budou prezentovány algoritmy pro odčítání. Při vypracovávání této podkapitoly byly použity prameny (2) a (3).

3.1.1 Jednobitové sčítačky

Neúplná jednobitová sčítačka (HA)

V literatuře bývá také označována jako poloviční sčítačka, což je doslovný překlad anglického označení HA (Half Adder). Představuje nejjednodušší realizaci sčítání, která neuvažuje vstupní přenos z nižšího řádu. Toto zapojení tedy nelze samostatně využít pro realizaci vícebitové sčítačky.

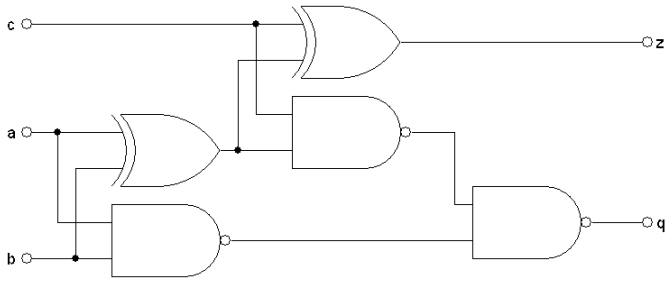
Výstupní funkce z, je realizována jako výlučný logický součet vstupů a a b, přenos do vyššího řádu c je realizován s využitím logického součinu. Schéma možné realizace neúplné jednobitové sčítačky zachycuje Obrázek 3.



Obrázek 3 - neúplná jednobitová sčítačka

Úplná jednobitová sčítačka (FA)

Označení FA je taktéž zkratkou anglického označení (Full Adder). Obvodová struktura je složitější než v případě HA. Jedná se v podstatě o kaskádní zapojení dvou neúplných jednobitových sčítaček, doplněné o hradlo AND, sloužící pro výpočet přenosu do vyššího řádu. Narozdíl od HA uvažuje tato struktura přenos z nižšího řádu, což je základní podmínka pro kaskádní realizaci vícebitové sčítačky. Obrázek 4 znázorňuje možnou realizaci struktury FA. Tabulka 1 představuje pravdivostní tabulku, vyjadřující logickou funkci obvodu.



Obrázek 4 - úplná jednobitová sčítačka

Tabulka 1 - pravdivostí tabulka úplné jednobitové sčítačky

a	b	c	z	q
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

3.1.2 Sčítačky s postupným výpočtem přenosu

Algoritmus Ripple-Carry

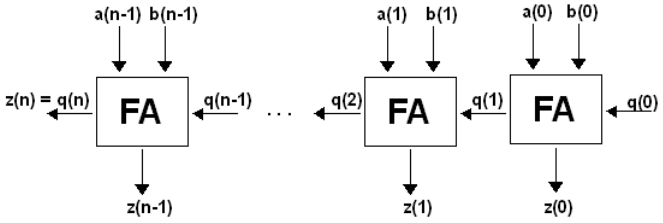
Sčítací algoritmus Ripple-Carry využívá kaskádního zapojení úplných jednobitových sčítaček. Jedná se o algoritmus vyznačující se jednoduchou strukturou, jehož předností je malá plocha potřebná pro realizaci. Blokové schéma sčítačky Ripple-Carry zachycuje

Obrázek 5. Výpočetní čas $T_{Ripple\text{-}Carry}$ a plocha obvodu $C_{Ripple\text{-}Carry}$ jsou přímo úměrně závislé na šířce vstupních operandů sčítačky *n* a lze je vypočítat podle vztahů:

$$T_{Ripple-Carry}\,(n) = \; n \cdot T_{FA} \tag{3.1}$$

$$C_{Ripple-Carry}\,(n) = \; n \cdot C_{FA} \tag{3.2}$$

C_{FA} a T_{FA} jsou plocha a výpočetní zpoždění úplné jednobitové sčítačky, která je použita pro vytvoření vícebitové struktury Ripple-Carry.



Obrázek 5 - blokové schéma sčítačky Ripple-Carry

3.1.3 Sčítačky s rychlým výpočtem přenosu

Postupný výpočet přenosu do vyššího řádu není kritickým pro krátké operandy. Budeme-li však zvyšovat délku operandů, nárůst výpočetního zpoždění bude neúnosný. Základní myšlenkou následujících algoritmů je urychlit právě výpočet přenosu a tím snížit výpočetní zpoždění. Logickým důsledkem této snahy je nárůst plochy nutné k implementaci takovýchto algoritmů. Blíže viz (2), nebo (3).

Algoritmus Carry-Chain

Algoritmus Carry-Chain je prvním ze skupiny algoritmů s rychlým přenosem. Jeho funkci lze rozdělit do tří úrovní, což dokumentuje jeho blokové schéma - Obrázek 6. V první úrovni reprezentované blokem G-P, dochází k výpočtu pomocných funkcí g (generate) a p (propagate). Funkce p signalizuje shodu vstupního a výstupního přenosu, funkce g potom nutnost ustavit výstupní přenos. Pro výpočet těchto funkcí slouží následující vztahy:

$$g(i) = \; a(i)\; and\; b(i) \tag{3.3}$$

$$p(i) = a(i)\; or\; b(i) \tag{3.4}$$

Výpočet pomocných funkcí v blocích G-P probíhá souběžně, na rozdíl od výpočtů v blocích CyCh. Tím dochází také k určité redukci výpočetního zpoždění.

Ve druhé úrovni, reprezentované blokem CyCh (Carry Chain), je generován přenos do vyššího řádu *q (i+1)*:

$$q(i+1) = g(i)\ or\ \big(p(i)\ and\ q(i)\big) \tag{3.5}$$

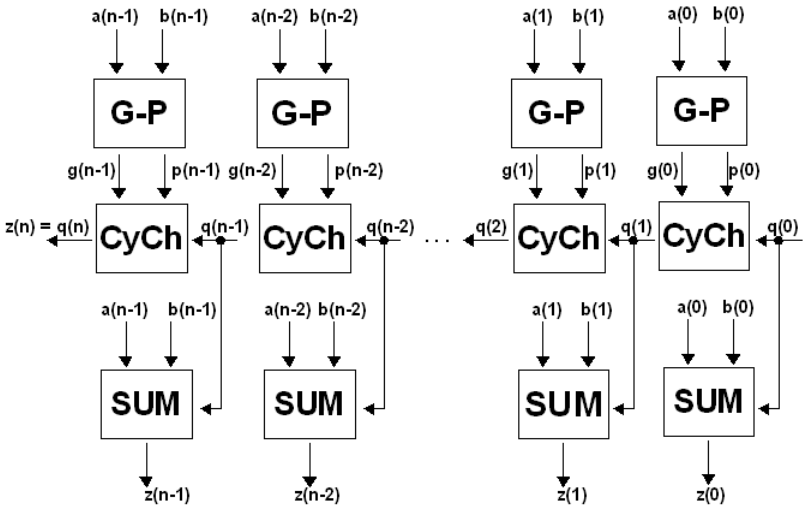
Třetí úroveň algoritmu představuje sumaci vstupních hodnot a přenosů. Realizována je blokem sumátoru SUM, jehož funkce lze popsat následujícím vztahem:

$$z(i) = x(i)\ xor\ y(i)\ xor\ c(i) \tag{3.6}$$

Výpočetní zpoždění této sčítačky lze získat jako součet zpoždění dílčích bloků. Nejprve dochází k výpočtu pomocných funkcí v bloku G-P, dále jsou vypočítávány přenosy do vyšších řádů (celkem *n*-1) a v posledním kroku dochází k sumaci výsledků. Tento proces je zachycen v rovnici (3.7). Plochu sčítačky lze vypočítat jako *n*-násobek součtu plochy dílčích bloků pomocí vztahu (3.8).

$$T_{Carry-Chain}\ (n) = T_{G-P} + \ (n-1) \cdot T_{CyCh} + T_{SUM} \tag{3.7}$$

$$C_{Carry-Chain}\ (n) = n \cdot (C_{G-P} + \ C_{CyCh} + C_{SUM}) \tag{3.8}$$



Obrázek 6 - blokové schéma sčítačky Carry-Chain

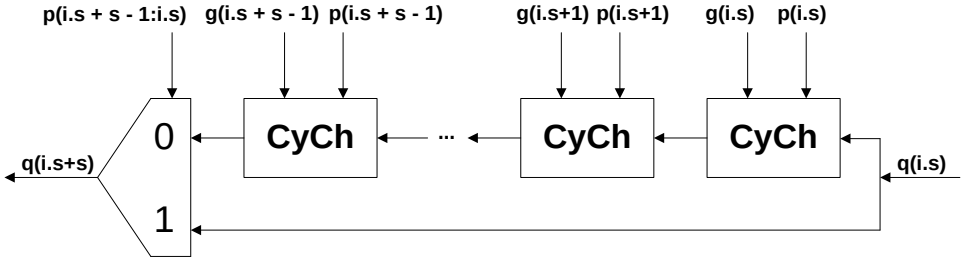
Algoritmus Carry-Skip

Algoritmus Carry-Skip rozděluje výpočet do několika podskupin o bitové šířce *s*. Dále dochází k testování hodnot pomocné funkce p pro všechny bity podskupiny. Za předpokladu, že hodnota všech bitů je rovna jedné (tuto podmínku lze v praxi velmi snadno testovat například pomocí hradla AND), výstupní přenos celé podskupiny odpovídá přenosu vstupnímu a není třeba dalších výpočtů. Tím dochází k redukci výpočetního zpoždění. Principiální schéma bloků výpočtu přenosu pro jednu podskupinu zachycuje

Obrázek 7. Výpočetní zpoždění tohoto algoritmu není konstantní a v nejhorším případě nedochází k žádné redukci. Maximální výpočetní zpoždění $T_{Carry\text{-}Skip}$ a plochu sčítačky $C_{Carry\text{-}Skip}$ lze stanovit podle následujících vztahů:

$$T_{Carry-Skip}\,(n) = T_{G-P} + \,(n-1)\cdot T_{Cy-Ch} + T_{SUM} \tag{3.9}$$

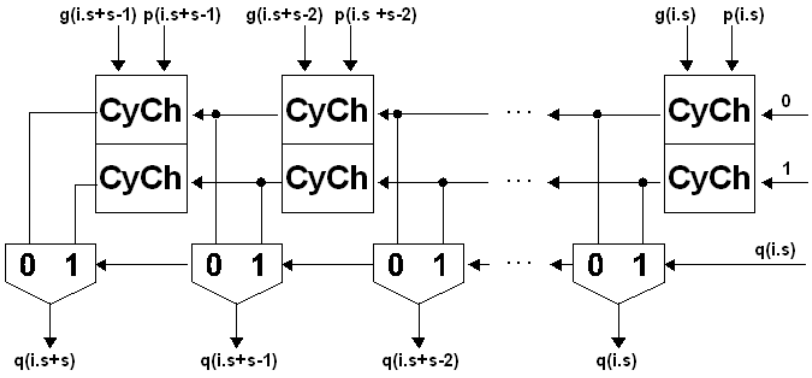
$$C_{Carry-Skip}\,(n) = n\cdot\left(C_{G-P} + \,C_{Cy-Ch} + C_{SUM}\right) + \left(\frac{n}{s}\right)\cdot\left(C_{MUX} + C_{ANDS}\right) \tag{3.10}$$



Obrázek 7 - blokové schéma sčítačky Carry-Skip – blok výpočtu přenosu

Algoritmus Carry-Select

Také algoritmus Carry-Select využívá rozdělení výpočtu do dílčích s-bitových podskupin. Na rozdíl od algoritmu Carry-Skip jsou však přenosy do vyšších řádů v bloku CyCh počítány vždy (s výjimkou první podskupiny) a to pro oba případy vstupního přenosu. Po stanovení výstupního přenosu předchozí podskupiny dochází, s využitím multiplexorů, k volbě odpovídající hodnoty výstupního přenosu každého bitu. Toto řešení vede vždy k redukci výpočetního zpoždění, které je navíc konstantní pro všechny hodnoty vstupních signálů. Jistým specifikem tohoto algoritmu je odlišná struktura první podskupiny, která je tvořena pouze bloky CyCh. Toto zjednodušení je umožněno díky znalosti vstupního přenosu již v době výpočtu hodnot přenosů. Blokové schéma výpočtu přenosu u sčítačky Carry-Select zachycuje Obrázek 8.



Obrázek 8 - blokové schéma sčítačky Carry-Select – blok výpočtu přenosu

Výpočetní zpoždění $T_{Carry\text{-}Select}$ a plochu struktury $C_{Carry\text{-}Select}$ stanovíme s využitím obdobné úvahy jako u předchozích struktur podle vztahů (3.11) a (3.12).

$$T_{Carry-Select}\,(n) = T_{G-P} + s \cdot T_{CyCh} + \left(\frac{n}{s} - 1\right) \cdot T_{Mux} + T_{Sum} \tag{3.11}$$

$$C_{Carry-Select}\,(n) = n \cdot \left(C_{G-P} + 2C_{CyCh} + C_{Mux} + C_{Sum}\right) - s \cdot \left(C_{CyCh} + C_{Mux}\right) \tag{3.12}$$

Algoritmus Carry Lookahead

Algoritmus Carry Lookahead lze chápat jako současný standard pro realizaci sčítaček. Ještě jednou je použito rozdělení výpočtu do několika podskupin a také tentokrát doznává změn blok výpočtu přenosu. Cílem této úpravy je vytvořit strukturu, která je schopná stanovit přenos každého bitu, bez znalosti přenosu bitu předchozího. Pro usnadnění popisu řešení je zavedena jednak matematická operace *dot*, jednak pojem kumulativní funkce *g* a *p*.

Mějme dva dvoubitové vektory $a_i = (a_{i0}; a_{i1})$ a $a_k = (a_{k0}; a_{k1})$. Matematickou operaci *dot* potom definujeme:

$$a_i\ dot\ a_k = (a_{i0}\ or\ a_{k0}\ and\ a_{i1}; a_{i1}\ and\ a_{k1}) \tag{3.13}$$

Dále uvažujme pomocné funkce *g(i)* a *p(i)* pro $i \in (0,1,\ldots\ n\text{-}1)$. Kumulativní funkce *g(i:i-k)* a *p(i:i-k)* pro $i \in (0,1,\ldots\ n\text{-}1)$ a $k \in (0,1,\ldots\ i)$ je definována:

$$\begin{aligned}&\left(g(i\colon i-k); p(i\colon i-k)\right) = \left(g(i), p(i)\right)\ dot\ \left(g(i-1), p(i-1)\right)\ dot\ \ \ldots\\ &\ldots\ dot\ \left(g(i-k), p(i-k)\right)\end{aligned} \tag{3.14}$$

Následující vztahy vyjadřují výpočet přenosu do vyšších řádů pro první tři bity. Z těchto vztahů je také patrný důvod zavedení operace *dot* a kumulativních funkcí *g* a *p*.

$$q_0 = c_{in} \tag{3.15}$$

$$q_1 = g_0\ or\ q_0\ and\ p_0 \tag{3.16}$$

$$q_2 = g_1\ or\ g_0\ and\ p_1\ or\ q_0\ and\ p_0\ and\ p_1 \tag{3.17}$$

Kumulativní funkce *g(i:i-k)* a *p(i:i-k)* nám umožňují stanovit přenos bez znalosti přenosu přecházejícího bitu. Tohoto principu využívá také blok CLA, jehož struktura je popsána následujícím kódem v jazyce VHDL.

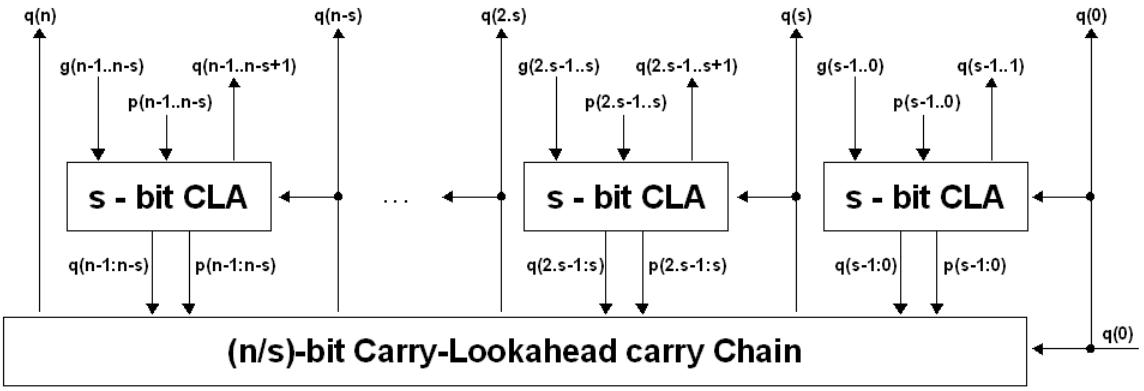
```
entity CLA is
generic(    gs:    integer);
port(c_in:  in     std_logic;
     g:     in     std_logic_vector(gs-1 downto 0);
     p:     in     std_logic_vector(gs-1 downto 0);
     g_g:   out    std_logic;
     g_p:   out    std_logic;
     q:     out    std_logic_vector(gs-1 downto 1));
end CLA;
architecture dataflow of CLA is
     signal ge_g:        std_logic_vector(gs-1 downto 0);
     signal ge_p:        std_logic_vector(gs-1 downto 0);
begin
     ge_g(0) <= g(0); ge_p(0) <= p(0);
     iteration: for i in 1 to gs-1 generate
            ge_g(i) <= g(i) or (ge_g(i-1) and p(i));
            ge_p(i) <= p(i) and ge_p(i-1);
            q(i)    <= ge_g(i-1) or (ge_p(i-1) and c_in);
     end generate;
     g_g <= ge_g(gs-1); g_p <= ge_p(gs-1);
end dataflow;
```

Další část bloku výpočtu přenosu tvoří, v případě sčítačky Carry-Lookahead, struktura (n/s)-bit Carry-Lookahead Carry Chain. Funkcí této struktury je výpočet vstupních přenosů pro bloky CLA a dále potom výpočet výstupního přenosu celé sčítačky. Ač se tato struktura může jevit velmi komplikovaně, v praxi bývá realizována dalším blokem CLA. Obrázek 9 zachycuje blok výpočtu přenosu sčítačky Carry-Lookahead.

Výpočetní zpoždění $T_{Carry\text{-}Lookahead}$ a plochu $C_{Carry\text{-}Lookahead}$ stanovíme podle vztahů:

$$T_{Carry-Lookahead}\,(n) = T_{G-P} + T_{dot}(s) + T_{cla}(n/s) + T_{carry} + T_{SUM} \tag{3.18}$$

$$C_{Carry-Lookahead}\,(n) = \left(\frac{n}{s}\right) \cdot \left(C_{dot}(s) + (s-1) \cdot C_{carry}\right) + C_{cla}(s/n) \tag{3.19}$$



Obrázek 9 - blokové schéma sčítačky Carry-Lookahead – blok výpočtu přenosu

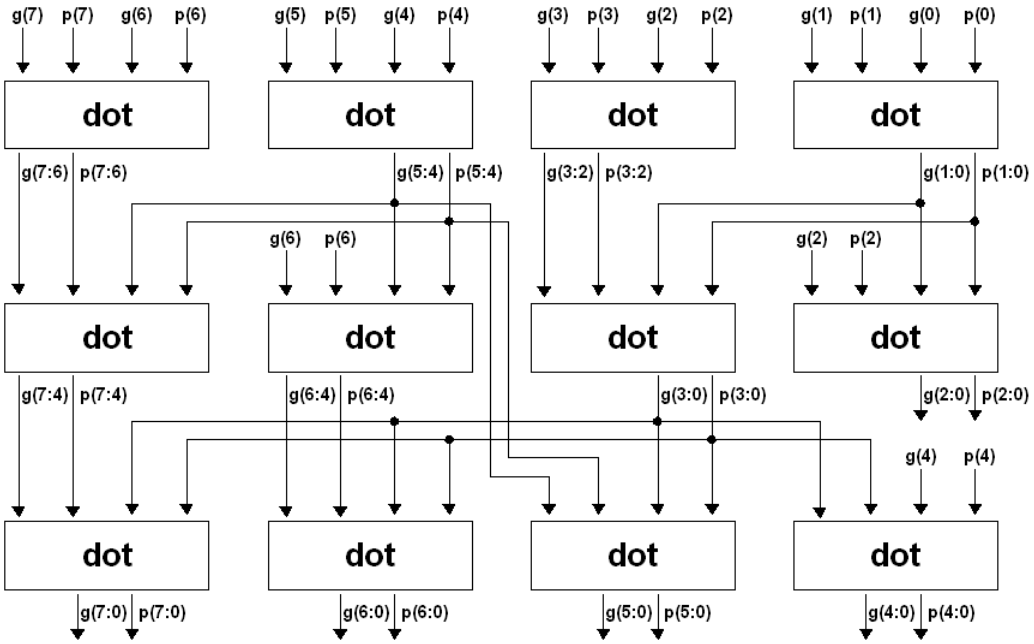
Algoritmus Brent-Kung

Algoritmus Brent-Kung bývá také označován jako Prefix Adder. Zachovává základní myšlenky algoritmu Carry-Lookahead, výpočet sám je však realizován v odlišné obvodové struktuře. Generátor přenosu využívá stromovou strukturu. Obrázek 10 zachycuje jeho blokovou strukturu.

Funkce generátoru lze rozdělit do dvou rovin. V první rovině dochází k výpočtu globálních kumulativní funkcí *p(n-1:0)* a *g(n-1:0)*. Souběžně s tímto výpočtem však také probíhá výpočet dílčích kumulativních funkcí *p(n-2:0)* až *p (n:0)* a *g(n-2:0)* až *g(1:0)*. Tato struktura je schopná provést kompletní výpočet v $log_2 n$ krocích, kde *n* je šířka vstupních slov sčítačky. Výpočetní zpoždění $T_{Brent\text{-}Kung}$ a plochu $C_{Brent\text{-}Kung}$ sčítačky stanovíme podle vztahů:

$$T_{Brent-Kung}(n) = T_{G-P} + (2 \cdot log_2 n - 2) \cdot T_{Dot} + T_{Carry} + T_{SUM} \quad (3.20)$$

$$C_{Brent-Kung}(n) = n \cdot (C_{G-P} + C_{Carry} + C_{SUM}) + (2 \cdot n - 2 - log_2 n) \cdot C_{Dot} \quad (3.21)$$



Obrázek 10 - blokové schéma 8 bitové sčítačky Brent-Kung – blok výpočtu přenosu

3.1.4 Odčítání

Struktury realizující odčítání jsou velmi podobné sčítačkám. Při návrhu digitálních obvodů obecně existuje snaha integrovat funkci specializovaných obvodových struktur do struktury víceúčelové. Tento přístup přináší vyšší efektivnost využití plochy cílového obvodu. Velmi často bývá uplatňován právě pro realizaci kombinované struktury pro sčítání a odčítání.

Odčítání čísel v doplňkovém kódu

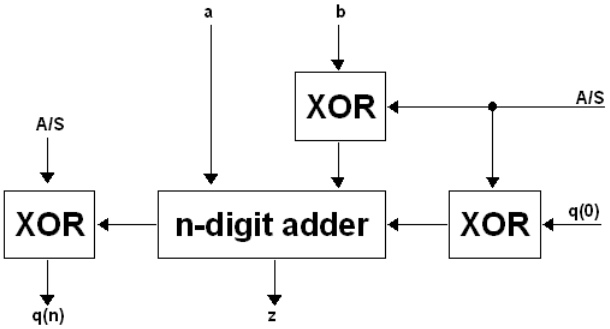
Pro realizaci odčítání čísel v doplňkovém kódu se využívá úvahy reprezentované rovnicí (3.22). Odčítání je převedeno na přičítání opačného čísla. Členy A a B představují vstupní operandy, členy +1 a –M jsou součástí negace čísla. Přičtení jedničky je realizováno s využitím vstupu přenosu q (0). Odečtení hodnoty M, představující modul sčítačky, není nutno realizovat, postačí po provedení operace zkontrolovat možnost odečtení. Výstupní přenos tedy musí být při správném průběhu odečítání roven jedné.

$$A - B = A + \bar{B} + 1 - M \tag{3.22}$$

Doplňkový kód však s sebou nese jedno riziko. Budeme-li sčítat dvě kladná čísla takových hodnot, že hodnota výsledku přesáhne kladný rozsah, bude výsledek nesprávně interpretován jako záporné číslo. Obdobné nebezpečí hrozí u záporných čísel, kde může být výsledek naopak interpretován jako kladné číslo. Pro indikaci těchto stavů postačí kontrolovat nejvyšší bity vstupních operandů, které v sobě nesou informaci o hodnotě znaménka. Tabulka 2 zachycuje možné stavy na vstupech a výstupu, červeně jsou vyznačeny stavy indikující přetečení. Tento problém lze snadno řešit přídavnou logikou sloužící k detekci nekorektních výsledků.

Tabulka 2 - Detekce nekorektních výstupních hodnot

MSB a	MSB b	MSB z	výskyt chyby
0	0	0	0
0	**0**	**1**	**1**
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	**1**	**0**	**1**
1	1	1	0



Obrázek 11 - blokové schéma odčítačky pro doplňkový kód

Blokové schéma kombinované struktury pro sčítání a odčítání v doplňkovém kódu zachycuje Obrázek 11. Kromě n-bitové sčítačky jsou zde využity bloky XOR, které slouží jako řízené invertory. Signál A/S představuje řídící signál, sloužící pro výběr sčítání (A/S = 0, signál prochází bloky XOR beze změny) nebo odčítání (A/S = 1, bloky XOR provádí negaci).

3.2 Násobení

Ve srovnání se sčítáním vyžaduje implementace násobení mnohem komplikovanější obvodové struktury. Logickým důsledkem je potom větší plocha cílového obvodu, nutná pro realizaci násobičky, a bohužel také větší výpočetní zpoždění.

Odpověď na otázku, zda realizovat popsané algoritmy sekvenčně či kombinačně, není v případě násobiček tak jednoznačná jak tomu bylo u sčítaček. V technické praxi však bývají preferovány struktury s nízkým výpočetním zpožděním. Právě z tohoto důvodu budou všechny níže popsané algoritmy realizovány jako kombinační obvody.

Při vypracovávání této podkapitoly bylo čerpáno z (2), (3) a (4).

3.2.1 Algoritmus Shift and Add

Anglický název Shift and Add, česky posuň a přičti, poměrně přesně vystihuje princip tohoto algoritmu. Ten vychází z klasické metody násobení s tužkou a papírem.

Výsledek je získán jako součet dílčích součinů násobence a a jednotlivých bitů násobitele b_0 - b_{nb-1}. Tyto bity také předávají dílčím součinům pozici bitu LSB. Bude-li mít operand a šířku n_a bitů a operand b šířku n_b bitů, výsledek bude šířky $n_z = (n_a + n_b - 1)$ bitů. Při výpočtu bude třeba n_a dílčích součinů o šířce n_b bitů. Obrázek 12 znázorňuje princip tohoto algoritmu. Algoritmus Shift and Add může být realizován s využitím n_b sčítaček o šířce n_a+n_b bitů. Jisté vylepšení lze dosáhnout úpravou mechanizmu sčítání, kdy při zachování počtu sčítaček dochází k redukci jejich šířky na n_a bitů.

					a_3	a_2	a_1	a_0	násobenec
x					b_3	b_2	b_1	b_0	násobitel
	0	0	0	0	a_3b_0	a_2b_0	a_1b_0	a_0b_0	
	0	0	0	a_3b_1	a_2b_1	a_1b_1	a_0b_1	0	
	0	0	a_3b_2	a_2b_2	a_1b_2	a_0b_2	0	0	
+	a_3b_3	a_3b_3	a_2b_3	a_1b_3	a_0b_3	0	0	0	
	y_7	y_6	y_5	y_4	y_3	y_2	y_1	y_0	výsledek

Obrázek 12 - princip algoritmu Shift and Add

3.2.2 Násobící pole

Jiný přístup přináší takzvaná násobící pole. Využívají zřetězení poměrně jednoduchých struktur, jejichž vnitřní struktura není závislá na šířce operandů. Šířka operandů naopak ovlivňuje rozměry násobícího pole

Funkci všech algoritmů této třídy lze rozdělit do dvou fází. V první z nich je prováděn výpočet dílčích výsledků systémem bit po bitu. Druhá fáze potom představuje sečtení těchto dílčích výsledků. Velmi obecný popis funkce násobícího pole determinuje širokou množinu možných realizací. To je dáno velkým počtem kombinací různých metod jak získat výsledky první či druhé fáze výpočtu.

Typickým příkladem realizace buňky násobícího pole je takzvaná základní násobící buňka. Její funkce zahrnuje první i druhou fázi výpočtu, a je popsána rovnicemi (3.23) a (3.24). Dále je přiložen výpis zdrojového kódu v jazyce VHDL, popisujícího tuto strukturu.

$$c_i\,(j+1) = \left(p_{i(i+j)} + x_i \cdot y_j + c_{ij}\right)/2 \tag{3.23}$$

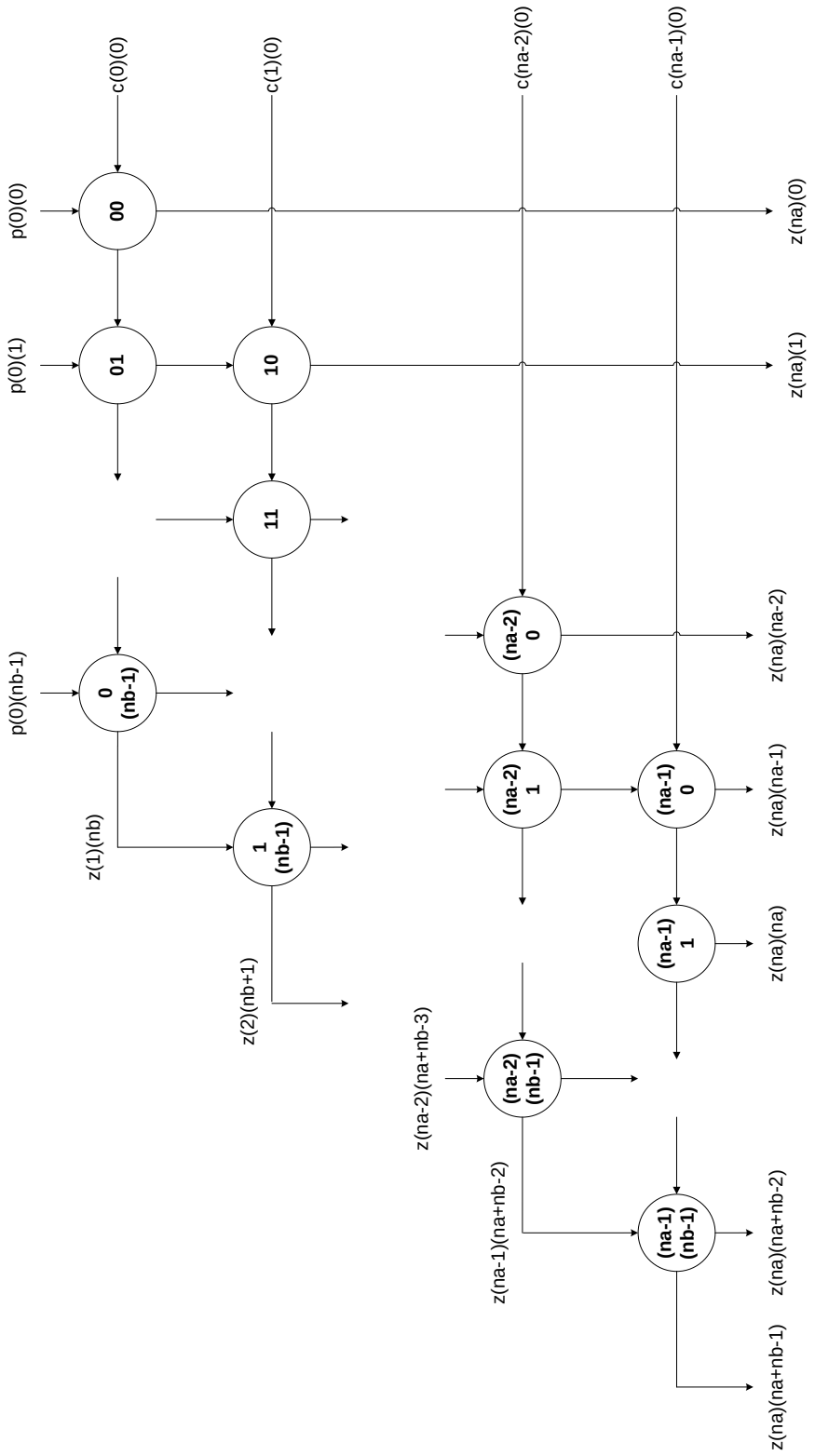
$$p_{(i+1)(i+j)} = \left(p_{i(i+j)} + x_i \cdot y_j + c_{ij}\right) \bmod 2 \tag{3.24}$$

```
entity Basic_Multiplier_Cell is
    port(
     a_i:          in    STD_LOGIC;
     b_j:          in    STD_LOGIC;
     c_in:         in    STD_LOGIC;
     p_in:         in    STD_LOGIC;
     c_out:        out   STD_LOGIC;
     p_out:        out   STD_LOGIC);
end Basic_Multiplier_Cell;

architecture Behavioral of Basic_Multiplier_Cell is
     signal ab: std_logic;
begin
     ab <= a_i and b_j;
     c_out <= (c_in and p_in) or (c_in and ab) or (p_in and ab);
     p_out <= c_in xor ab xor p_in;
end Behavioral;
```

Násobící pole Ripple-Carry

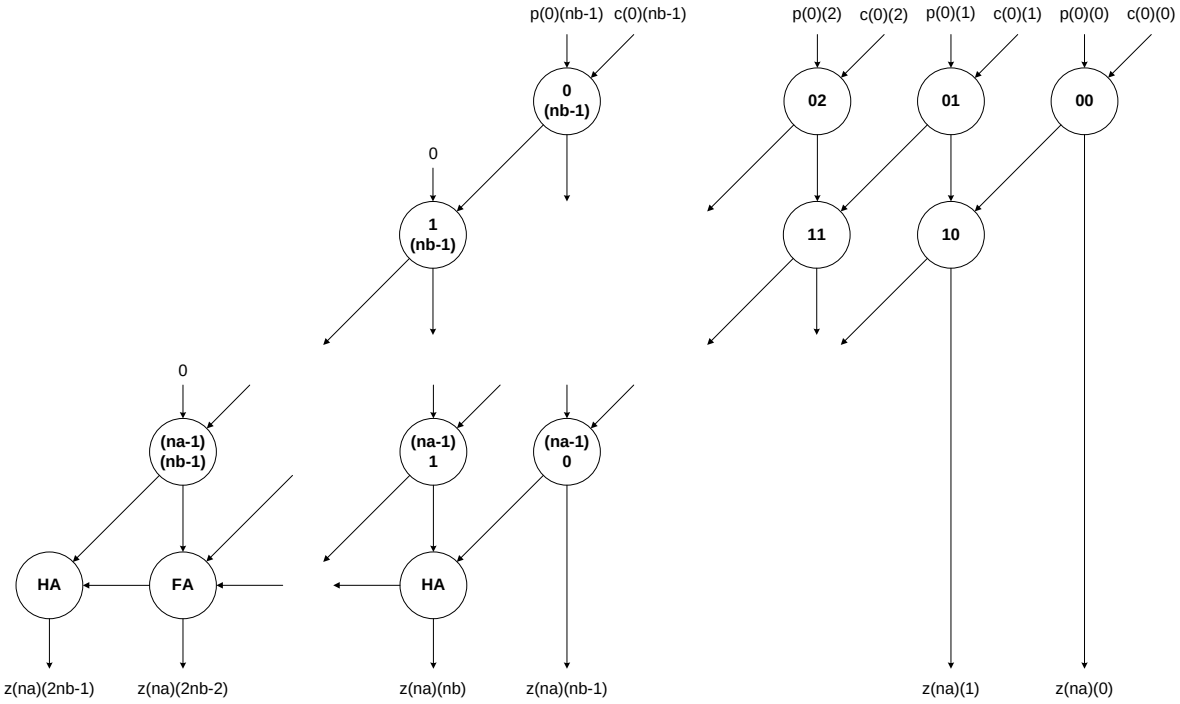
Tento algoritmus představuje spojení základních násobících buněk do struktury, jež zachycuje Obrázek 13. Každý řádek představuje n_b*-1* bitovou sčítačku Ripple Carry. Jeden vstupní operand tvoří výsledek vyššího stupně označený *p* a druhý výsledek je logický součin příslušných bitů vstupních operandů *a, b*. Vstupní přenos c_{x0} je nulový.



Obrázek 13 - blokové schéma násobícího pole Ripple-Carry

Násobící pole Carry-Save

Násobička Carry-Save využívá poněkud odlišnou strukturu násobícího pole (viz Obrázek 14). Také zde je však využita základní násobící buňka. Struktura vychází z principu multi-operandové sčítačky s uchováním přenosu označované anglickým termínem Carry-Save. Odtud také název. Jak je patrné z blokových schémat, přináší toto řešení oproti variantě Ripple-Carry redukci plochy obvodu. Nižší počet členů se dále projevuje také nižším výpočetním zpožděním.



Obrázek 14 - blokové schéma násobícího pole Carry-Save

3.2.3 Algoritmus Booth

Boothův algoritmus patří mezi nejefektivnější způsoby implementace násobení do digitálních obvodů. Jeho funkce je založena na využití soustavy relativních číslic. Na rozdíl od binární číselné soustavy, tato soustava obsahuje tři číslice: 0, 1, -1. Zavedení třetího znaku umožňuje odlišný a v mnohém efektivnější způsob vyjadřování čísel. Například číslo 15 je v binární reprezentaci vyjádřeno jako 00001111. Při využití Boothova překódování dostáváme následující vyjádření: 0001000-1 v dekadické soustavě reprezentované takto: 16-1.

Princip tohoto překódování je uplatňován na všechny skupiny jedniček (za skupinu považujeme i samostatnou jedničku). Překódování je možné provádět s využitím převodní tabulky. Pokud budeme využívat dvou vstupních bitů, hovoříme o překódování s radixem 2. Tabulka 3 zachycuje toto překódování.

Tabulka 3 - překódování do Boothova kódu s radixem 2

Číslice na pozici i	Číslice na pozici i+1	Boothův kód na pozici i
0	0	0
0	1	1
1	0	-1
1	1	0

Takovéto překódování je výhodné zejména pro záporná čísla. V technické praxi se často využívá překódování s vyšším radixem. Radix 4 využívá 3 bitů, radix 8 potom 4 bitů. Takováto úprava vede k vyšší rychlosti překódování a i nižšímu výpočetnímu zpoždění.

3.3 Dělení

Dělení představuje vzhledem k realizaci v digitálním obvodu nejkomplikovanější z elementárních matematických operací. Při realizaci matematických struktur je preferováno sčítání a odčítání. Časté je také násobení. Naopak dělení je využíváno v omezené míře.

Významnou roli také hraje fakt, že množina celých čísel je oproti sčítání, odčítání a násobení uzavřená, ne však oproti dělení. Jinak řečeno součet, rozdíl, nebo součin dvou celých čísel je opět celé číslo. V případě dělení tento předpoklad nemusí platit. Další informace o děličkách viz (2).

3.3.1 Dělení přirozených čísel

Mějme dělenec *A* a dělitel *B*, pro které platí $A > 0$ a $B > 0$. Dále definujme podíl těchto čísel *Z* a zbytek po dělení *R* tak, že platí:

$$A = Z \cdot B + R \qquad kde\ 0 \leq |R| \leq B \tag{3.25}$$

Výpočet hodnoty podílu je u následujících algoritmů prováděn postupně. Proto zavedeme ještě průběžný zbytek R_i a i-tý bit podílu Z_i. Pro stanovení hodnoty i-tého bitu podílu lze uplatnit následující podmínky:

Je-li 2^{-i} B menší než R_i, pak $Z_i = 1$

Je-li 2^{-i} B větší než R_i, pak $Z_i = 0$

nový průběžný zbytek vypočteme jako:

$$R_{i+1} = R_i - Z_i \cdot 2^{-i} \cdot B \tag{3.26}$$

v praxi se však častěji využívá vztah:

$$R_{i+1} = 2R_i - Z_i \cdot B \tag{3.27}$$

nyní máme k dispozici dvě možnosti, jak realizovat dělící algoritmus:

Dělička s návratem k nenulové hodnotě

Tato struktura v případě záporné hodnoty nového průběžného zbytku provede znovu přičtení odečtené hodnoty.

Dělička bez návratu k nenulové hodnotě

Naopak tato struktura v případě záporného zbytku převede v následujícím kroku odečítání na přičítání. V případě tohoto algoritmu je vždy po ukončení výpočtu nutné kontrolovat hodnotu zbytku R, a pokud by byl záporný, provést korekci.

3.3.2 Dělička SRT

Název tohoto algoritmu je akronymem jmen jeho tvůrců (Sweeney, Robertson, Tocher). Při své funkci využívá soustavy relativních číslic jako je tomu u Boothova algoritmu. Hodnota jednotlivých bitů podílu, je přitom odhadnuta s využitím náhledové tabulky z několika nejvyšších bitů průběžného zbytku R_i.

3.4 Goniometrické funkce

Do skupiny základních matematických operací můžeme kromě sčítání, odčítání, násobení a dělení zařadit také goniometrické funkce sinus, kosinus a tangens. Právě jejich implementaci je věnována tato kapitola. Obdobě jako u předchozích skupin matematických operací i zde existuje více způsobů, kterými lze výpočet goniometrických funkcí realizovat. Následující podkapitoly popisují dva z těchto způsobů.

3.4.1 Aproximace goniometrických funkcí Taylorovým polynomem

Goniometrické funkce lze vyjádřit jako mocninnou řadu. K tomuto účelu nám výborně poslouží Taylorova řada. Rovnice (3.28) a (3.29) vyjadřují Taylorovu řadu pro funkce sinus a kosinus. Z těchto rovnic také vyplývá, že pro získání přesné hodnoty je zapotřebí nekonečného počtu prvků.

V technické praxi však není nezbytné využívat zcela přesných hodnot. V naprosté většině případů je možné, velmi často je dokonce účelné, použít přibližnou (aproximovanou) hodnotu s postačující přesností. Při využití vztahů popsaných rovnicemi (3.30) a (3.31)

je relativní chyba menší než 0,5 %. Pro zvýšení přesnosti potom postačí zvýšit počet členů posloupnosti, které jsou využity při výpočtu. Blíže viz (5).

$$sin(x) = \sum_{n=0}^{\infty}(-1)^n \frac{x^{2n+1}}{(2n+1)!} \qquad pro\ x\ \in (-\infty, \infty) \tag{3.28}$$

$$cos(x) = \sum_{n=0}^{\infty}(-1)^n \frac{x^{2n}}{(2n)!} \qquad pro\ x\ \in (-\infty, \infty) \tag{3.29}$$

$$sin(x) = \frac{x^1}{1!} - \frac{x^3}{3!} + \frac{x^5}{5!} \qquad pro\ x\ \in (-\infty, \infty) \tag{3.30}$$

$$cos(x) = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} \qquad pro\ x\ \in (-\infty, \infty) \tag{3.31}$$

$$tg(x) = x + \frac{x^3}{3} + \frac{2x^5}{15} \qquad pro\ x\ \in \left(-\frac{\pi}{2}, \frac{\pi}{2}\right) \tag{3.32}$$

Jak je z předchozího textu zřejmé, jedná se o velmi jednoduchou metodu pro výpočet hodnot goniometrických funkcí. Pro implementaci do digitálních obvodů je však překážkou nutnost opakovaného dělení.

3.4.2 Algoritmus CORDIC

Zcela jiný přístup k výpočtu hodnot goniometrických funkcí přináší algoritmus CORDIC (Coordinate Rotation Digital Computer). Jedná se o iterační algoritmus, který pro svoji funkci využívá pouze sčítání, odčítání a bitové posuvy.

Podle pramene (6) je rotaci jakéhokoli vektoru o úhel φ možné vyjádřit podle vztahů:

$$x_\mathrm{r} = x_0 \cos(\varphi) - \mathrm{y}_0 \sin(\varphi) \tag{3.33}$$

$$y_\mathrm{r} = x_0 \sin(\varphi) + y_0 \cos(\varphi) \tag{3.34}$$

Pro účely dalšího výpočtu je vhodné tyto vztahy upravit:

$$\frac{x_\mathrm{r}}{\cos(\varphi)} = x_0 - \mathrm{y}_0 \,\mathrm{tg}(\varphi) \tag{3.35}$$

$$\frac{y_\mathrm{r}}{\cos(\varphi)} = \mathrm{y}_0 + x_0 \,\mathrm{tg}(\varphi) \tag{3.36}$$

Druhou úpravou potom dostaneme:

$$x_\mathrm{r} = \cos(\varphi)\,(x_0 - \mathrm{y}_0 \,\mathrm{tg}(\varphi)) \tag{3.37}$$

$$y_\mathrm{r} = \cos(\varphi)\,(\mathrm{y}_0 + x_0 \,\mathrm{tg}(\varphi)) \tag{3.38}$$

Pokud budeme volit úhel φ takovým, aby hodnota jeho tangenty tohoto úhlu odpovídala hodnotě 2^{-i} pro i > 0, je možné tangentu ve vzorci nahradit právě zápornou mocninou čísla dvě. Její výpočet lze snadno realizovat pomocí bitového posuvu. Takto stanovené hodnoty tangenty úhlu dále vymezují přesné úhly, o které je možné rotaci provádět. Nezáleží však na směru rotace a proto můžeme výsledný úhel získat vhodnou sérií rotací. Další úvaha spočívá v nahrazení rotace parciální rotací. Hodnoty cos φ lze potom nahradit konstantou K_i. Dostáváme vztahy:

$$x_r = K_i(x_0 - y_0 d_i\, 2^{-1}) \tag{3.39}$$

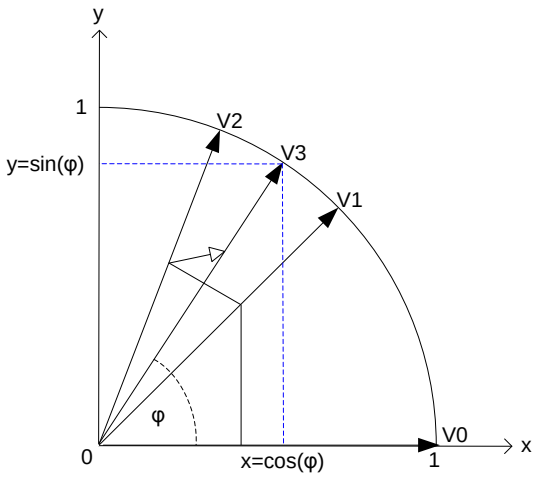
$$y_r = K_i(y_0 + x_0\, d_i\, 2^{-1}) \tag{3.40}$$

kde d_i vyjadřuje směr rotace (tento parametr nabývá pouze hodnot -1 a 1). Hodnota konstanty K_i lze stanovit podle vztahu:

$$K_i = \cos(\operatorname{arctg} 2^{-i}) = 1/\left(1 + 2^{-2i}\right)^{1/2} \tag{3.41}$$

Limitně se tato hodnota blíží ke K_i = 0,6073. Protože při parciální rotaci se pohybujeme za hranicí jednotkové kružnice, výsledné hodnoty jsou zesíleny. Toto zesílení je vyjádřeno konstantou K_i a pro korekci výstupních hodnot postačí tyto hodnoty konstantou K_i vydělit.

Obsluha algoritmu CORDIC spočívá pouze ve správném nastavení počátečních souřadnic. Pro výpočet goniometrických funkcí sinus a kosinus je využívána kladná reálná poloosa. Po průchodu daného počtu rotací a korekci hodnot, odpovídají průměty vektoru v osách x a y hodnotám kosinu a sinu zadaného úhlu. Obrázek 15 ilustruje průběh prvních tří iterací algoritmu CORDIC.



Obrázek 15 - průběh iterací algoritmu CORDIC

4 Realizace zvolených algoritmů

V předchozí kapitole byly popsány funkční principy dvacítky zvolených algoritmů, vyhovujících pro implementaci matematických operací do digitálních obvodů FPGA či ASIC. Tato kapitola se věnuje jednak konkrétnímu způsobu jejich realizace (popisu ve VHDL), jednak provedení syntézy těchto algoritmů.

4.1 Popis algoritmů jazykem VHDL

Popis algoritmů byl realizován s využitím generických proměnných. Generické proměnné umožňují nastavit některé parametry popisované struktury prostřednictvím hodnoty známé v okamžiku syntézy. V tomto případě byly generické proměnné využity pro určení šířky operandů a v případě sčítaček pro optimalizační nastavení.

Je zcela samozřejmé, že byla ověřena funkčnost vzniklých zdrojových kódů. Pro tento účel byl využit simulační program ModelSim XE III 6.3c od společnosti Mentor Graphics.

Ověřené kódy byly sdruženy do implementačních knihoven (balíčků), rozdělených podle realizované funkce. Vytvoření těchto knihoven bylo jedním z hlavních cílů projektu.

Sčítání

Struktura označená **AGA** představuje implementaci sčítání vytvořenou syntetizátorem. Tento algoritmus byl zařazen z důvodu ověření vlastností takto implementovaných struktur. Konkrétně je tato struktura popsána následujícím kódem v jazyce VHDL:

```
architecture Behavioral of AGA is
begin
     z <= a+b;
end Behavioral;
```

Algoritmy **Ripple-Carry** a **Carry-Chain** byly realizovány s využitím jedné generické proměnné n, kterou je určena jak šířka obou vstupních operandů, tak šířka výsledku.

Algoritmy **Carry-Skip**, **Carry-Select** a **Carry-Lookahead** využívají taktéž generickou proměnnou *n*, navíc však využívají generickou proměnnou *s*, která slouží pro popis vnitřní struktury. Přesný význam této proměnné je nastavení počtu podskupin, do kterých je rozdělen výpočet (viz výše).

Také algoritmus **Brent-Kung** využívá dvou generických proměnných. První z nich je opět *n*, sloužící pro určení šířky operandů a výsledků, druhá z nich je nazvaná *lvl* a vyjadřuje počet úrovní generátoru přenosu. Hodnota proměnné *lvl* je závislá na proměnné *n* podle vztahu $lvl = log_2\ n$. Logaritmická funkce bohužel není v jazyce VHDL implementována, proto je třeba hodnotu proměnné *lvl* nastavovat ručně.

Odčítání

Struktura **AGS** představuje implementaci odčítání generovanou syntetizátorem. Tato struktura je v jazyce VHDL popsána následujícím kódem:

```
architecture Behavioral of AGS is
begin
     z <= a+b;
end Behavioral;
```

Struktura **AGAS** je kombinovanou strukturu pro sčítání a odčítání realizovanou s využitím sčítačky a odčítačky automaticky vytvořené syntetizátorem. Pro její popis byl využit následující kód v jazyce VHDL:

```
architecture Behavioral of AGAS is
begin
     z <= a - b when control = '1' else a+b;
end Behavioral;
```

Struktura **SCAS** představuje spojení doplňkové logiky a jednoho z realizovaných sčítacích algoritmů. Výsledkem je kombinovaná struktura pro sčítání a odčítáni čísel v doplňkovém kódu. Jazyk VHDL bohužel neumožňuje podmíněný překlad, požadovaný sčítací algoritmus je tedy zvolen odstraněním značek komentáře od názvu příslušné komponenty v entitě algoritmu SCAS. Tento algoritmus využívá dvě generické proměnné. Proměnná *n* slouží jednak pro vytvoření pomocné kombinační logiky, jednak společně s proměnnou s určují parametry použité sčítačky. Následuje popis algoritmu v jazyce VHDL.

```
architecture behavioral of SC_AdderSubtracters is
     signal qt:    std_logic; -- q temporary
     signal ct:    std_logic; -- c temporary
     signal bt:    std_logic_vector (n-1 downto 0); -- b temporary
     signal zt:    std_logic_vector (n-1 downto 0); -- z temporary
begin
     ct <= c_in xor control;
     iter: for i in 0 to n-1 generate bt(i) <= b(i) xor control;
     end generate;
     adder_n: Ripple_Carry_Adder
            generic map  (n)
            port map     (ct,qt,a,bt,zt);
--                 .
--                 .
--                 .
--   adder_n: BrentKung_Adder
--          generic map  (n,s)
--          port map     (ct,qt,a,bt,zt);
     ovf <= (not(a(n-1))and not(bt(n-1)) and zt(n-1)) or
            (a(n-1) and bt(n-1) and not(zt(n-1)));
     c_out <= qt xor control;
     z <= zt;
end behavioral;
```

Násobení

Struktura AGM opět představuje implementaci realizovanou automaticky syntetizátorem. Struktura je popsána následujícím kódem v jazyce VHDL:

```
architecture Behavioral of AGM is
begin
     z <= a*b;
end Behavioral;
```

Algoritmy **Shift and Add, Ripple-Carry, Carry-Save, Booth radix 2, Booth radix 4 a Booth radix 8** využívají tři generické proměnné. Proměnná *na* slouží k nastavení šířky operandu a, proměnná *nb* slouží k nastavení šířky operandu b a proměnná *nz* slouží k nastavení šířky výstupu. Tyto proměnné by měly být provázány vztahem $na + nb = nz$.

Dělení

Struktury děliček **RTZ**, **NRTZ** a **STR radix 2** využívají dvou generických proměnných. První z nich je proměnná *n*, která slouží k nastavení šířky dělence *a*, dělitele *b* a zbytku po dělení *r*. Druhou je proměnná *np*, sloužící pro nastavení šířky podílu *z*. Všechny výše zmíněné struktury jsou určeny pro operandy ve zlomkovém tvaru. Dělitel by tedy měl být větší než dělenec, přičemž váha MSB výsledku by měla být poloviční než váha LSB vstupních operandů.

Goniometrické funkce

Pro realizaci výpočtu goniometrických funkcí byla vytvořena struktura **SinCosCordic**. Tato struktura je realizována sekvenčně, přičemž počet iterací určuje přesnost výsledku.

K nastavení počtu iterací slouží generická proměnná *cnt_iter*. Šířka výstupních i vstupních dat je fixní: 32 bitů. Vstupním parametrem této funkce je úhel, pro který hledáme hodnotu funkce sinus, nebo kosinus. Výběr této funkce provádíme prostřednictvím řídícího signálu fcf. Vzhledem k využitému číselnému formátu, je vstupní úhel třeba znásobit konstantou $k = 10^6$. Tato úprava představuje v dekadické soustavě posun o šest řádů, čímž je zajištěna poměrně vysoká přesnost tohoto algoritmu.

4.2 Syntéza algoritmů

Dalším krokem je provedení syntézy získaných zdrojových kódů. Právě během procesu syntézy jsou získávána data o parametrech výsledných realizací. Tato data jsou zatížena jistou chybou, protože během syntézy nejsou uvažovány parazitní jevy, které se projevují po implementaci algoritmu do konkrétního obvodu. Protože působení parazitních jevů uvnitř

cílového obvodu lze pouze odhadnout, výsledky syntézy potom představují ideální referenční podmínky. Díky této úvaze lze vzniklou nepřesnost zanedbat.

Z názvu práce vyplývá, že cílovým obvodem je FPGA. Z množiny dostupných obvodů byl zvolen Spartan-3 XC3S200 od firmy Xilinx. Právě tento obvod je osazován do vývojové desky Spartan-3 Starter kit, použité během práce na tomto projektu. Pro syntézu zdrojových kódů byl použit syntetizátor integrovaný ve vývojovém prostředí Xilinx ISE 11.1, jak název napovídá, taktéž dodávaný firmou Xilinx.

V průběhu práce na projektu bylo rozhodnuto o přidání druhého cílového obvodu. Vedoucím práce byl zvolen obvod ASIC realizovaný technologií AMIS 350 nm. Zde byl pro syntézu využit nástroj Cadence Encounter RTL Compiler verze 07.10-s009_1 od firmy Cadence.

Výše popsané vývojové nástroje umožňují poměrně široké uživatelské nastavení procesu syntézy. Bližší informace o možnostech syntetizátoru Cadence Encounter RTL Compiler viz (7). Pro účel této práce byly uvažovány pouze dva typické případy, tedy snaha o dosažení minimálního výpočetního zpoždění a snaha o dosažení minimální plochy realizovaného obvodu. Tyto dvě optimalizace syntézy jsou ve výsledcích označeny jako optimalizace-rychlost a optimalizace-plocha.

5 Srovnání popsaných algoritmů

Posledním z cílů této práce je zhodnotit realizované algoritmy a porovnat je s realizacemi, které vytváří syntetizátory. Algoritmy budou posuzovány na základě informací o jejich výpočetním zpoždění a ploše, která je nutná pro jejich realizaci. V případě obvodů ASIC byla získána také data, která nám přibližují spotřebu těchto obvodů.

Během práce na projektu byl získán poměrně rozsáhlý soubor dat. Z důvodu zachování přehlednosti tohoto textu budou v následujících podkapitolách uvedeny výsledky struktur s délkou vstupních operandů 16 bitů. Kompletní soubor dat je přiložen na konci této práce jako Příloha 1 až Příloha 24. Vyhodnocení algoritmů je obdobně jako v předchozích kapitolách rozděleno podle matematické funkce realizované obvodem.

5.1 Algoritmy pro sčítání a odčítání

Implementace sčítání a potažmo odčítání je v případě architektury FPGA velmi kvalitně provedena algoritmy AGA a AGS. Ty představují struktury, které jsou automaticky generovány syntetizátorem při zjištění zástupného znaku + nebo - ve zdrojovém kódu. Takto realizované struktury dosahují nejen nižšího výpočetního zpoždění, ale také výrazně nižší plochy nutné k realizaci struktury.

Ačkoli tato možnost nebyla v době teoretické přípravy předpokládána, zdůvodnění tohoto jevu je poměrně snadné. Vnitřní struktura obvodů FPGA je tvořena mimo jiné elementárními buňkami (slices). Tyto buňky slouží k realizaci obvodové funkce. Mimo náhledové tabulky a klopného obvodu obsahují také pomocnou aritmetickou logiku. Tento fakt je doložen v publikaci (8). Zatímco při tvorbě algoritmů AGA a AGS je tato pomocná logika využita, při implementaci ostatních logických funkcí využita není.

Právě využití pomocné aritmetické logiky vede k redukci plochy výsledného obvodu. Nepřímo se potom projevuje možností efektivnějšího využití lokálních propojovacích polí, což vede ke snížení výpočetního zpoždění.

Jiná situace nastala při implementaci realizovaných algoritmů do architektury ASIC. Byl potvrzen význam sčítacích struktur využívajících generátory rychlého přenosu. Toto tvrzení lze dokumentovat výsledky algoritmu Brent-Kung, jehož výpočetní zpoždění bylo ve srovnání s implementací AGA méně než poloviční. Cenou za toto snížení výpočetního zpoždění bylo poměrně značné navýšení plochy obvodu. Plocha nutná pro implementaci sčítačky Brent-Kung je přibližně třikrát větší, než v případě algoritmu AGA. Výše prezentované závěry dokumentuje Tabulka 4 a Tabulka 5.

Tabulka 4 - výpočetní zpoždění sčítaček - šířka operandů 16 bitů

Algoritmus	Výpočetní zpoždění [ps]			
	ASIC – optimalizace syntézy		FPGA – optimalizace syntézy	
	Rychlost	Plocha	Rychlost	Plocha
AGA	5 841	7 232	9 758	9 758
Ripple Carry	6 045	7 516	13 131	28 741
Carry Chain	4 194	7 516	13 131	28 741
Carry Skip (s=2)	5 219	7 747	13 204	27 853
Carry Skip (s=4)	5 207	6 612	17 806	36 037
Carry Skip (s=8)	5 201	6 010	14 465	30 142
Carry Select (s=2)	8 158	5 825	13 131	27 853
Carry Select (s=4)	8 158	3 912	13 131	29 172
Carry Select (s=8)	8 158	3 692	12 835	21 325
Carry Lookahead (s=2)	3 098	3 202	13 089	22 562
Carry Lookahead (s=4)	2 962	2 928	14 960	18 578
Carry Lookahead (s=8)	3 581	3 538	13 151	19 618
BrentKung	2 446	2 447	13 064	21 471

Tabulka 5 - plocha sčítaček - šířka operandů 16 bitů

Algoritmus	Plocha struktury [ekviv. hradel]		Plocha struktury [slices]	
	ASIC – optimalizace syntézy		FPGA – optimalizace syntézy	
	Rychlost	Plocha	Rychlost	Plocha
AGA	72	72	8	8
Ripple Carry	75	75	32	18
Carry Chain	99	75	32	18
Carry Skip (s=2)	133	176	35	18
Carry Skip (s=4)	133	165	33	25
Carry Skip (s=8)	133	163	36	25
Carry Select (s=2)	107	163	32	18
Carry Select (s=4)	107	192	32	18
Carry Select (s=8)	107	199	47	29
Carry Lookahead (s=2)	151	151	40	35
Carry Lookahead (s=4)	150	151	38	33
Carry Lookahead (s=8)	151	151	54	38
BrentKung	198	198	43	26

Pro implementaci odčítání může být kromě již zmíněného algoritmu AGS využit algoritmus AGAS. Ten spadá taktéž do skupiny algoritmů vytvářených automaticky syntetizátorem. Na rozdíl od algoritmu AGS však představuje kombinovanou strukturu pro sčítání i odčítání.

Poslední možnost jak implementovat odčítání čísel v doplňkovém kódu je algoritmus SCAS. Jedná se o typickou realizaci kombinované struktury pro sčítání a odčítání ve dvojkovém kódu.

Jeho funkce je založena na některém ze sčítacích algoritmů popsaných v rámci této práce. Vzhledem k poměrně komplexnímu přehledu parametrů realizovaných sčítacích algoritmů, byla při syntéze vyextrahována pouze nástavbová část a k hodnotám uvedeným u tohoto algoritmu je tedy nutné přičíst hodnoty použité sčítačky.

Parametry implementovaných algoritmů zachycuje Tabulka 6 a Tabulka 7.

Tabulka 6 - výpočetní zpoždění odčítaček - šířka operandů 16 bitů

Algoritmus	Výpočetní zpoždění [ps]			
	ASIC – optimalizace syntézy		FPGA – optimalizace syntézy	
	Rychlost	Plocha	Rychlost	Plocha
AGS	5 859	7 253	9 758	9 758
AGAS	6 233	7 562	9 984	9 984
SCAS	595	712	8 186	8 311

Tabulka 7 - plocha odčítaček - šířka operandů 16 bitů

Algoritmus	Plocha struktury [ekviv. hradel]		Plocha struktury [slices]	
	ASIC – optimalizace syntézy		FPGA – optimalizace syntézy	
	Rychlost	Plocha	Rychlost	Plocha
AGS	82	82	8	8
AGAS	106	187	8	8
SCAS	40	40	11	11

5.2 Algoritmy pro násobení

Situace při implementaci násobiček do obvodů FPGA je velmi podobná situaci vzniklé při implementaci sčítaček. Tentokrát není důvodem omezený přístup ke struktuře obvodu, ale využití specializovaných hardwarových struktur uvnitř architektury FPGA. Jejich počet je omezen, nicméně díky výborným parametrů představují významný přínos pro snížení výpočetního zpoždění realizované struktury.

Při implementaci násobení do architektury ASIC není již tolik výrazná převaha struktur realizovaných v této práci, ačkoli nejlepších výsledků dosahuje násobící pole Carry-Save. Pravděpodobná příčina tohoto propadu je nedokonalá optimalizace realizovaných struktur. Vzhledem k primárnímu zacílení práce na architekturu FPGA je například sčítání realizováno algoritmem AGA, který vykazuje nejlepší vlastnosti pro FPGA. U architektury ASIC by mohl

být nahrazen například algoritmem Brent-kung, který ovšem povede k dalšímu navýšení plochy realizovaného obvodu.

Přehled získaných výsledků obsahuje Tabulka 8 a Tabulka 9.

Tabulka 8 - výpočetní zpoždění násobiček - šířka operandů 16 bitů

Algoritmus	Výpočetní zpoždění [ps]			
	ASIC – optimalizace syntézy		FPGA – optimalizace syntézy	
	Rychlost	Plocha	Rychlost	Plocha
AGM	13 896	16 576	11 207	11207
Shift And Add	19 325	22 098	45 840	47590
Ripple Carry	19 300	19 325	60 242	73665
Carry Save	13 453	15 374	29 574	33319
Booth radix 2	26 493	33 394	66 132	71084
Booth radix 4	22 443	27 742	39 090	42359
Booth radix 8	27 052	31 137	35 166	38825

Tabulka 9 - plocha násobiček - šířka operandů 16 bitů

Algoritmus	Plocha struktury [ekviv. hradel]		Plocha struktury [slices]	
	ASIC – optimalizace syntézy		FPGA – optimalizace syntézy	
	Rychlost	Plocha	Rychlost	Plocha
AGM *)	1 265	1 279	0	0
Shift And Add	1 435	1 435	249	121
Ripple Carry	1 435	1 435	357	293
Carry Save	1 435	1 435	274	276
Booth radix 2	2 289	3 401	393	309
Booth radix 4	2 700	3 319	226	228
Booth radix 8	4 494	5 088	500	479

5.3 Algoritmy pro dělení

V rámci této práce byly realizovány tři základní algoritmy pro implementaci dělení. Vzhledem k předpokládaným parametrům těchto algoritmů představují výsledky implementace do obvodů FGPA úspěch.

Na druhou stranu je opravdu paradoxní, že výpočetní zpoždění pro stejnou obvodovou strukturu je u architektury ASIC vyšší než u architektury FPGA. Možnou příčinou je opět nedokonalá optimalizace implementovaných struktur. Tabulka 10 a Tabulka 11 zachycují výsledné parametry implementovaných algoritmů,

Tabulka 10 - výpočetní zpoždění násobiček - šířka operandů 16 bitů

Algoritmus	Výpočetní zpoždění [ps]			
	ASIC – optimalizace syntézy		FPGA – optimalizace syntézy	
	Rychlost	Plocha	Rychlost	Plocha
RTZ	117 675	144 759	92 570	92 570
NRTZ	127 842	79 604	75 487	75 487
SRT radix 2	130 380	146 800	78 829	78 829

Tabulka 11 - plocha děliček - šířka operandů 16 bitů

Algoritmus	Plocha struktury [ekviv. hradel]		Plocha struktury [slices]	
	ASIC – optimalizace syntézy		FPGA – optimalizace syntézy	
	Rychlost	Plocha	Rychlost	Plocha
RTZ	1 835	1 995	283	283
NRTZ	1 790	3 099	161	161
SRT radix 2	2 217	3 596	162	162

5.4 Algoritmus CORDIC

Všechny výše uvedené algoritmy byly realizovány jako kombinační struktury. Algoritmus CORCIC je však realizován sekvenčně. Výpočetní zpoždění tohoto algoritmu je definováno jako *(n + 2)* násobek hodinového taktu. Parametr *n* reprezentuje počet iterací nutných k dosažení výsledku se zadanou přesností. Další dva strojové cykly jsou nutné pro zajištění správného vstupu a výstupu dat.

Druhým aspektem, který je nutno ve spojitosti se sekvenčními obvody sledovat, je maximální kmitočet při kterém může daná obvodová struktura pracovat. Ten se při implementaci do architektury FPGA pohybuje v intervalu 70,715 ÷ 74,862 MHz. Pro implementaci potom potřebuje v závislosti na volené optimalizaci 439 ÷ 464 slices.

Ověření možnosti implementace do architektury ASIC bohužel nebylo možné. I přes opakované pokusy o odstranění problému, skončily všechny syntézy již během procesu elaborace chybou.

6 Realizace kalkulačky s využitím FPGA

Během práce na tomto projektu vyvstala potřeba ověřit vytvořené algoritmy nejen pomocí testovacích programů, ale také prakticky. Po konzultaci s vedoucím práce bylo rozhodnuto otestovat popsané algoritmy po implementaci do obvodů FPGA. Pro tyto účely byl v jazyce VHDL vytvořen popis kalkulačky využívající algoritmy realizované v rámci tohoto projektu

Pro realizaci kalkulačky byl využit Spartan-3 Starter kit osazený obvodem FPGA Spartan-3 XC3S200. Tato vývojová deska bohužel neobsahuje numerickou klávesnici, ani displej o dostatečném počtu znaků, tyto prvky tedy museli být doplněny prostřednictvím rozšiřujících modulů. Použity byly moduly LDC displeje a maticové klávesnice vytvořené na Ústavu mikroelektroniky. Celkový pohled na modul kalkulačky je zachycuje Obrázek 16.

Obrázek 16 - fotka: Spartan-3 kalkulačka – celkový pohled

Jádro kalkulačky tvoří stavový automat. Po resetu, případně vynulováním kalkulačky (tlačítko *), je stavový automat uveden do výchozího stavu a vyčkává na zadání prvního operandu. Tento stav ilustruje Obrázek 17 a.

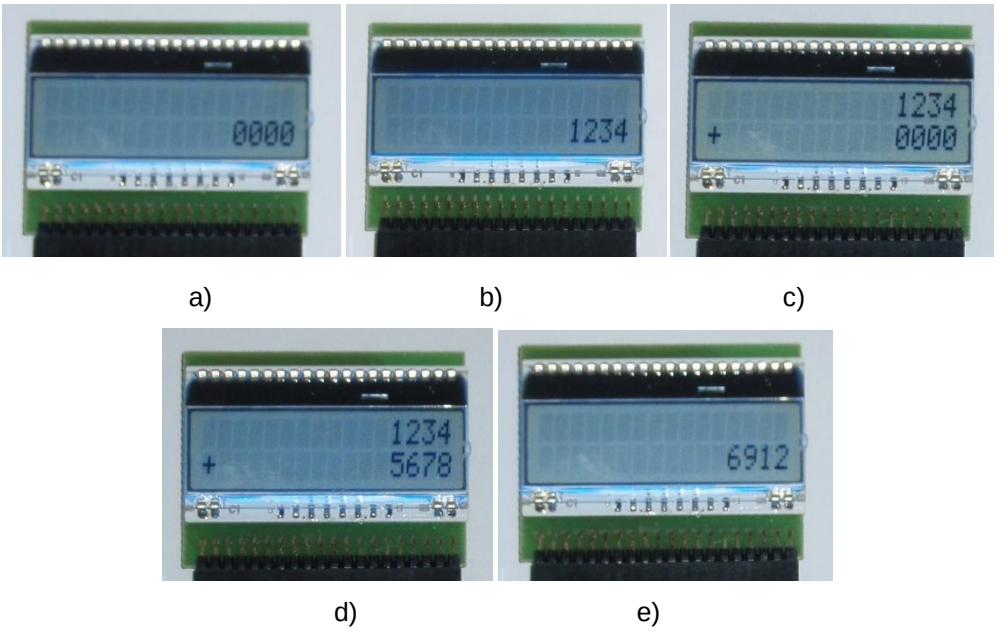
Vzhledem k omezené ploše cílového obvodu, byla zvolena šestnáctibitová délka vstupních operandů pro všechny realizované matematické operace. Šestnáctibitové číslo může vyjádřit 65 536 kombinací, tato kalkulačka pracuje se 4 BCD číslicemi, čímž je vyloučeno přetečení při sčítání odčítání a dělení. U násobení je třeba volit velikost operandu respektující maximální hodnotu výsledku.

Pro zadání operandu lze použít tlačítka 0 až 9. Vstupní operand je uchováván uvnitř posuvného BCD registru s šířkou 16 bitů. Počet zadaných číslic je omezen na 4. Stav po zadání prvního operandu ilustruje Obrázek 17 b.

Tlačítka A, B, C a D slouží k volbě matematické operace v pořadí sčítání, odčítání, násobení, dělení. Stiskem jednoho s těchto tlačítek dojde k výběru prováděné matematické operace a vyvolání konverze BCD čísla ze vstupního registru na binární číslo. Toto binární číslo je přivedeno jako první operand do implementovaných matematických struktur. Dále dochází k vynulování vstupního registru a čekání na druhý operand. Tento stav zachycuje Obrázek 17 c. Opět je pomocí tlačítek 0 až 9 možné zadat druhý operand. Tuto situaci ilustruje Obrázek 17 d.

Po zadání druhého operandu mohou nastat dvě situace. První z nich je stisk tlačítka #, které zastupuje rovnítko. Dojde k převodu vstupního operandu na binární číslo, je započat výpočet, výsledek je převeden na BCD číslo a zobrazen na displeji (viz Obrázek 17 e). Druhou variantou je stisk některého z tlačítek zastupujících matematickou operaci. V tomto případě je vykonána stejná posloupnost operací, výsledek je však chápán jako první operand a kalkulačka očekává vložení druhého operandu. Tento stav zachycuje Obrázek 17 c.

Zdrojové kódy této kalkulačky jsou součástí obsahu přiloženého disku DVD.



a) b) c)

d) e)

a - připravena k zadání prvního operandu; b - po zadání prvního operandu;
c - po zvolení matematické operace; d - po zadání 2. operandu; e - zobrazení výsledku

Obrázek 17 - fotka: Spartan-3 kalkulačka – detail displeje

7 Zhodnocení

Tato práce byla zaměřena na návrh a realizaci matematických operací v prostředí digitálních obvodů. Jako cílová technologie byl zvolen obvody FPGA Spartan-3. Práce na tomto projektu byla rozdělena do několika dílčích částí.

Cílem první částí projektu bylo zvolit vhodný číselný formát, který bude při realizaci jednotlivých algoritmů využit. Volba padla na formát pevné řádové čárky, přičemž pro vyjádření záporných čísel byl zvolen doplňkový kód.

Druhá část byla zaměřena na návrh a realizaci vhodných algoritmů pro implementaci sčítání, respektive odčítání. Význam této části je dán především skutečností, že sčítačky a odčítačky jsou využívány při implementaci složitějších matematických operací. Zvolené algoritmy měli podle prvotních předpokladů kvalitativně překonávat automaticky generované struktury, výsledky syntéz však tyto předpoklady nepotvrdily. Pro implementaci sčítání a odčítání do algoritmů realizovaných v následujících částech projektu byly zvoleny struktury automaticky generované syntetizátorem.

Ve třetí části projektu byly vybrány a realizovány struktury pro implementaci násobení. Zároveň byl také přidán druhý cílový obvod. Zvolen byl obvod ASIC, realizovaný technologií AMIS 350 nm. V rámci této části byly také vyhodnoceny parametry již realizovaných algoritmů při implementaci do druhého cílového obvodu.

Čtvrtá část byla věnována implementaci dělení a goniometrických funkcí. Zanedbáno nebylo ani vyhodnocení parametrů realizovaných struktur při implementaci do obou cílových obvodů.

Cílem páté části projektu bylo vytvoření kalkulačky, která by umožňovala ověření funkce algoritmů pro sčítání, odčítání, násobení a dělení.

Šestá část byla věnována vypracování této zprávy, která dokumentuje celý projekt, realizovaný v rámci bakalářské práce.

V dalším pokračování projektu lze předpokládat implementaci matematických operací pro čísla ve formátu plovoucí řádové čárky, implementaci výpočtu mocnin a odmocnin, případně implementaci dalších goniometrických funkcí (arkussinus, arkuskosinus,…).

8 Závěr

V rámci této práce byly vybrány vhodné algoritmy pro realizaci matematických operací v prostředí digitálních obvodů. Tyto algoritmy byly popsány jazykem VHDL a analyzovány. Popisy jednotlivých algoritmů byly sdruženy podle realizované matematické operace, čímž vznikly implementační knihovny. Díky přenositelnosti jazyka VHDL, bude možné tyto knihovny budoucnu využít jak pro obvody FPGA, tak pro obvody ASIC. Zdrojové kódy těchto knihoven jsou součástí obsahu přiloženého disku DVD.

V průběhu práce byla dále získána data o výpočetním zpoždění a ceně takto realizovaných struktur v prostředí FPGA i ASIC. Vyhodnocení těchto dat přineslo několik velmi zajímavých závěrů.

Implementace matematických operací sčítání, odčítání a násobení do obvodů FPGA je při využití realizovaných algoritmů méně efektivní než při využití struktur generovaných syntetizátorem. Z tohoto důvodu nelze využití těchto algoritmů doporučit.

Při implementaci sčítání, odčítání a násobení do obvodů ASIC dochází k potvrzení předpokladů spojených s konkrétními algoritmy. Využitím komplikovanější obvodové struktury jsme schopni několikanásobně snížit výpočetní zpoždění ve srovnání se strukturou vytvořenou syntetizátorem. Pro implementaci těchto operací je tedy možné doporučit algoritmy popsané v rámci této práce.

V případě implementace dělení a výpočtu goniometrických funkcí do obou typů obvodů, není syntetizátor schopen automaticky vygenerovat požadovanou strukturu. Řešení proto představují algoritmy popsané v rámci této práce. Další možnost realizace těchto operací spočívá ve využití příslušných ip core, kde ovšem může při komerčním využití aplikace dojít ke střetu s autorskými právy.

Během realizace projektu byla s využitím vývojové desky Spartan-3 Starter kit vytvořena kalkulačka, která slouží pro ověření implementovaných algoritmů.

9 Seznam použitých zdrojů

1. **PINKER, Jiří, POUPA, Martin.** *Číslicové systémy a jazyk VHDL.* Praha : BEN, 2006. str. 352. ISBN 80-7300-198-5.

2. **DESCHAMPS, Jean-Pierre, BIOUL, Géry Jean Antonie a SUTTER, Gustavo D.** *Synthesis of Arithmetic Circuits: FPGA, ASIC and Embedded Systems.* New Jersey : Wiley, 2006. str. 556. ISBN 978-0-471-68783-2.

3. **LU, Mi.** *Arithmetic and logic in computer systems.* New Jersey : Wiley, 2004. str. 246. ISBN 0-471-46945-9.

4. **PARHAMI, Behrooz.** *Computer Arithmetic: Algorithms and Hardware Designs.* New York : Oxford University Press, 2000. str. 490. ISBN 0-19-512583-5.

5. **TIŠNOVSKÝ, Pavel.** Aritmetické operace s hodnotami ve formátu plovoucí řádové čárky. *Root.cz.* [Online] [Citace: 7. 3 2009.] <http://www.root.cz/clanky/aritmeticke-operace-s-hodnotami-ve-formatu-plovouci-radove-carky>.

6. **TIŠNOVSKÝ, Pavel.** Výpočet goniometrických funkcí algoritmem CORDIC. *Root.cz.* [Online] [Citace: 14. 3 2009.] <http://www.root.cz/clanky/vypocet-goniometrickych-funkci-algoritmem-cordic>.

7. *Encounter RTL Compiler V7.2.* Lecture manual : Cadence, 7. March 2008.

8. *Spartan-3 FPGA Family Data Sheet.* [elektronický dokument] DS099 June 25, 2008, Xilinx : Product specification. <http://www.xilinx.com>.

10 Seznam použitých zkratek a symbolů

AGA Obvodová struktura sčítačky generovaná syntetizátorem - označení použité v rámci této práce

AGAS Obvodová struktura kombinující funkci sčítačky a odčítačky generovaná syntetizátorem - označení použité v rámci této práce

AGM Obvodová struktura násobičky generovaná syntetizátorem - označení použité v rámci této práce

AGS Obvodová struktura odčítačky generovaná syntetizátorem - označení použité v rámci této práce

ASIC Application specific integrated circuit - zákaznický obvod

CLA Carry-Lookahead

CyCh Carry-Chain

dot Matematická funkce zavedená v tomto textu. Přináší zjednodušení popisu obvodové funkce u sčítaček s rychlým výpočtem přenosu

DSP Digitální signálový procesor

FA Full adder - úplná jednobitová sčítačka

FP Floating point - formát plovoucí řádové čárky

FPGA Field programmable gate array - programovatelné hradlové pole

FX Fixed point - formát pevné řádové čárky

G-P Blok výpočtu pomocných funkcí g a p

HA Half adder - neúplná jednobitová sčítačka

NRTZ Non return to zero - princip realizující dělení bez návratu k nenulové hodnotě

RTZ Return to zero - princip realizující dělení s návratem k nenulové hodnotě

SCAS Obvodová struktura odčítačky pro doplňkový kód - označení použité v rámci této práce

SRT Sweeney, Robertson a Tocher - princip realizující dělení v oboru celých čísel

VHDL Very high speed integrated circuits hardware description language - jazyk pro popis hardwaru

11 Seznam příloh

12 Obsah disku DVD

Složka A – Elektronická verze této práce

Složka B – Zdrojové kódy sčítaček a odčítaček

Složka C – Zdrojové kódy násobiček

Složka D – Zdrojové kódy děliček

Složka E – Zdrojové kódy algoritmu CORDIC

Složka F – Zdrojové kódy ověřovací implementace - Spartan-3 kalkulačka

13 Přílohy

Příloha 1 - parametry sčítaček, FPGA implementace, optimalizace syntézy: rychlost

Algoritmus	Výpočetní zpoždění [ps]			Plocha struktury [slices]		
	8 bitů	16 bitů	32 bitů	8 bitů	16 bitů	32 bitů
AGA	9 314	9 758	10 646	4	8	16
Ripple-Carry	11 036	13 131	14 720	15	32	75
Carry-Chain	11 036	13 131	14 720	15	32	75
Carry-Skip (s=2)	11 636	13 204	14 739	14	35	76
Carry-Skip (s=4)	12 464	17 806	28 654	16	33	68
Carry-Skip (s=8)	-	14 465	18 570	-	36	81
Carry-Skip (s=16)	-	-	16 009	-	-	79
Carry-Select (s=2)	11 036	13 131	14 720	15	32	75
Carry-Select (s=4)	11 036	13 131	14 720	15	32	75
Carry-Select (s=8)	-	12 835	14 521	-	47	88
Carry-Select (s=16)	-	-	16 426	-	-	82
Carry-Lookahead (s=2)	11 911	13 089	14 405	20	40	102
Carry-Lookahead (s=4)	11 029	14 960	15 647	20	38	97
Carry-Lookahead (s=8)	-	13 151	15 588	-	54	115
Carry-Lookahead (s=16)	-	-	15 016	-	-	127
Brent-Kung	10 705	13 064	14 721	19	43	98

Příloha 2 - parametry sčítaček, FPGA implementace, optimalizace syntézy: rychlost

Algoritmus	Výpočetní zpoždění [ps]			Plocha struktury [slices]		
	8 bitů	16 bitů	32 bitů	8 bitů	16 bitů	32 bitů
AGS	9 314	9 758	10 646	4	8	16
AGAS	9 352	9 984	11 309	4	8	16
SCAS *)	7 926	8 186	8 544	6	11	20

Příloha 3 - parametry sčítaček, FPGA implementace, optimalizace syntézy: plocha

Algoritmus	Výpočetní zpoždění [ps]			Plocha struktury [slices]		
	8 bitů	16 bitů	32 bitů	8 bitů	16 bitů	32 bitů
AGA	9 314	9 758	10 646	4	8	16
Ripple-Carry	17 585	28 741	51 052	9	18	37
Carry-Chain	17 585	28 741	51 052	9	18	37
Carry-Skip (s=2)	17 627	27 853	49 276	9	18	37
Carry-Skip (s=4)	21 171	36 037	62 895	13	25	51
Carry-Skip (s=8)	-	30 142	53 974	-	25	50
Carry-Skip (s=16)	-	-	52 564	-	-	46
Carry-Select (s=2)	17 627	27 853	49 276	9	18	37
Carry-Select (s=4)	17 738	29 172	50 164	9	18	37
Carry-Select (s=8)	-	21 325	31 102	-	29	59
Carry-Select (s=16)	-	-	25 948	-	-	59
Carry-Lookahead (s=2)	13 789	22 562	23 488	13	35	64
Carry-Lookahead (s=4)	13 629	18 578	25 757	12	33	70
Carry-Lookahead (s=8)	-	19 618	23 783	-	38	78
Carry-Lookahead (s=16)	-	-	31 051	-	-	79
Brent-Kung	15 052	21 471	26 620	11	26	58

Příloha 4 - parametry sčítaček, FPGA implementace, optimalizace syntézy: plocha

Algoritmus	Výpočetní zpoždění [ps]			Plocha struktury [slices]		
	8 bitů	16 bitů	32 bitů	8 bitů	16 bitů	32 bitů
AGS	9 314	9 758	10 646	4	8	16
AGAS	9 352	9 984	11 309	4	8	16
SCAS *)	8 051	8 311	8 669	6	11	20

Příloha 5 - parametry sčítaček, ASIC implementace, optimalizace syntézy: rychlost

Algoritmus	Výpočetní zpoždění [ps]			Plocha struktury [ekviv. hradel]		
	8 bitů	16 bitů	32 bitů	8 bitů	16 bitů	32 bitů
AGA	2 825	5 841	11 873	35	72	147
Ripple-Carry	3 029	6 045	12 077	37	75	149
Carry-Chain	2 459	4 194	7 738	49	99	197
Carry-Skip (s=2)	2 814	5 219	10 048	67	133	267
Carry-Skip (s=4)	2 808	5 207	10 005	67	133	227
Carry-Skip (s=8)	-	5 201	9 992	-	133	227
Carry-Skip (s=16)	-	-	9 986	-	-	227
Carry-Select (s=2)	3 993	8 158	16 487	53	107	213
Carry-Select (s=4)	3 993	8 158	16 487	53	107	213
Carry-Select (s=8)	-	8 158	16 487	-	107	213
Carry-Select (s=16)	-	-	16 487	-	-	213
Carry-Lookahead (s=2)	2 080	3 098	5 198	74	151	306
Carry-Lookahead (s=4)	2 420	2 962	4 012	73	150	305
Carry-Lookahead (s=8)	-	3 581	4 123	-	151	305
Carry-Lookahead (s=16)	-	-	5 846	-	-	305
Brent-Kung	1 933	2 446	3 242	87	198	443

Příloha 6 - parametry sčítaček, ASIC implementace, optimalizace syntézy: rychlost

Algoritmus	Výpočetní zpoždění [ps]			Plocha struktury [ekviv. hradel]		
	8 bitů	16 bitů	32 bitů	8 bitů	16 bitů	32 bitů
AGS	2 843	5 859	11 891	39	82	167
AGAS	3 217	6 233	12 265	53	106	213
SCAS *)	595	595	595	24	40	72

Příloha 7 - parametry sčítaček, ASIC implementace, optimalizace syntézy: plocha

Algoritmus	Výpočetní zpoždění [ps]			Plocha struktury [ekviv. hradel]		
	8 bitů	16 bitů	32 bitů	8 bitů	16 bitů	32 bitů
AGA	3 434	7 232	14 827	35	72	147
Ripple-Carry	3 718	7 516	15 111	37	75	149
Carry-Chain	2 785	7 516	8 572	55	75	219
Carry-Skip (s=2)	4 123	7 747	14 996	88	176	352
Carry-Skip (s=4)	3 559	6 612	12 717	83	165	331
Carry-Skip (s=8)	-	6 010	11 544	-	163	327
Carry-Skip (s=16)	-	-	11 002	-	-	321
Carry-Select (s=2)	3 199	5 825	11 077	81	163	325
Carry-Select (s=4)	2 472	3 912	6 091	96	192	384
Carry-Select (s=8)	-	3 692	5 985	-	199	399
Carry-Select (s=16)	-	-	5 847	-	-	410
Carry-Lookahead (s=2)	2 152	3 202	5 303	74	151	306
Carry-Lookahead (s=4)	2 386	2 928	3 978	74	151	306
Carry-Lookahead (s=8)	-	3 538	4 080	-	151	305
Carry-Lookahead (s=16)	-	-	5 842	-	-	304
Brent-Kung	1 960	2 447	3 139	87	198	443

Příloha 8 - parametry sčítaček, ASIC implementace, optimalizace syntézy: plocha

Algoritmus	Výpočetní zpoždění [ps]			Plocha struktury [ekviv. hradel]		
	8 bitů	16 bitů	32 bitů	8 bitů	16 bitů	32 bitů
AGS	3 456	7 253	14 848	39	82	167
AGAS	3 765	7 562	15 157	91	187	379
SCAS *)	712	712	712	24	40	72

Příloha 9 - spotřeba sčítaček, ASIC implementace, optimalizace syntézy: rychlost

Algoritmus	Klidová spotřeba [nW]			Dynamická spotřeba [µW]		
	8 bitů	16 bitů	32 bitů	8 bitů	16 bitů	32 bitů
AGA	9,36	19,74	40,49	277,11	472,15	994,04
Ripple-Carry	10,38	20,75	41,51	237,64	521,70	1 042,95
Carry-Chain	14,24	27,88	55,17	294,57	628,28	1 260,20
Carry-Skip (s=2)	15,66	31,32	62,64	292,94	628,30	1 262,53
Carry-Skip (s=4)	15,06	30,12	60,23	291,76	625,68	1 257,97
Carry-Skip (s=8)	-	29,51	59,03	-	624,45	1 255,67
Carry-Skip (s=16)	-	-	58,43	-	-	1 254,45
Carry-Select (s=2)	13,31	26,62	53,25	243,22	526,84	1 061,07
Carry-Select (s=4)	13,31	26,62	53,25	243,22	526,83	1 061,07
Carry-Select (s=8)	-	26,62	53,25	-	526,83	1 061,07
Carry-Select (s=16)	-	-	53,25	-	-	1 061,07
Carry-Lookahead (s=2)	15,92	31,91	63,89	362,88	762,99	1 518,11
Carry-Lookahead (s=4)	16,36	32,35	64,33	356,90	785,84	1 548,40
Carry-Lookahead (s=8)	-	33,54	65,52	-	790,41	1 522,08
Carry-Lookahead (s=16)	-	-	67,49	-	-	1 528,87
Brent-Kung	16,69	36,12	77,81	423,35	969,92	2 026,23

Příloha 10 - spotřeba sčítaček, ASIC implementace, optimalizace syntézy: rychlost

Algoritmus	Klidová spotřeba [nW]			Dynamická spotřeba [µW]		
	8 bitů	16 bitů	32 bitů	8 bitů	16 bitů	32 bitů
AGS	9,66	20,12	41,05	270,78	552,15	1 235,64
AGAS	16,60	33,48	67,24	422,25	844,45	1 787,13
SCAS *)	9,03	15,53	28,54	60,38	94,20	152,16

Příloha 11 - spotřeba sčítaček, ASIC implementace, optimalizace syntézy: plocha

Algoritmus	Klidová spotřeba [nW]			Dynamická spotřeba [µW]		
	8 bitů	16 bitů	32 bitů	8 bitů	16 bitů	32 bitů
AGA	9,36	19,74	40,49	213,55	519,21	1 418,11
Ripple-Carry	10,38	20,75	41,51	260,89	576,39	1 156,62
Carry-Chain	15,61	20,75	62,43	325,01	576,39	1 396,97
Carry-Skip (s=2)	16,29	32,58	65,16	423,50	906,71	1 813,78
Carry-Skip (s=4)	16,36	32,73	65,46	383,63	827,67	1 657,07
Carry-Skip (s=8)	-	31,38	62,77	-	767,77	1 551,31
Carry-Skip (s=16)	-	-	64,47	-	-	1 562,91
Carry-Select (s=2)	17,62	35,24	70,48	366,59	799,17	1 613,01
Carry-Select (s=4)	22,15	44,31	88,63	457,25	1 009,83	2 050,04
Carry-Select (s=8)	-	46,65	93,31	-	1 113,48	2 284,95
Carry-Select (s=16)	-	-	95,63	-	-	2 384,18
Carry-Lookahead (s=2)	16,22	32,21	64,19	369,14	766,14	1 526,89
Carry-Lookahead (s=4)	16,67	32,66	64,64	366,80	792,03	1 561,61
Carry-Lookahead (s=8)	-	33,82	65,80	-	782,67	1 526,27
Carry-Lookahead (s=16)	-	-	68,11	-	-	1 560,41
Brent-Kung	16,69	36,12	77,81	422,71	977,51	2 035,94

Příloha 12 - spotřeba sčítaček, ASIC implementace, optimalizace syntézy: plocha

Algoritmus	Výpočetní zpoždění [ps]			Plocha struktury [slices]		
	8 bitů	16 bitů	32 bitů	8 bitů	16 bitů	32 bitů
AGS	9,66	20,12	41,05	280,12	570,93	1 289,03
AGAS	21,43	44,67	91,14	669,38	1 386,34	2 999,70
SCAS *)	8,98	15,48	28,48	59,37	92,37	155,06

Příloha 13 - parametry násobiček, FPGA implementace, optimalizace syntézy: rychlost

Algoritmus	Výpočetní zpoždění [ps]			Plocha struktury [slices]		
	8 bitů	16 bitů	32 bitů	8 bitů	16 bitů	32 bitů
AGM	10 132	11 207	17 213	0	0	40
Shift And Add	24 859	45 840	88 065	56	249	1 049
Ripple-Carry	29 646	60 242	121 121	87	357	1 268
Carry Save	18 547	29 574	50 456	65	274	1 101
Booth radix 2	36 455	66 132	126 936	130	393	1 805
Booth radix 4	23 049	39 090	71 317	61	226	854
Booth radix 8	22 814	35 166	56 449	111	500	1 872

Příloha 14 - parametry násobiček, FPGA implementace, optimalizace syntézy: plocha

Algoritmus	Výpočetní zpoždění [ps]			Plocha struktury [slices]		
	8 bitů	16 bitů	32 bitů	8 bitů	16 bitů	32 bitů
AGM	10 132	11 207	17 338	0	0	40
Shift And Add	25 609	47 590	91 815	29	121	497
Ripple-Carry	36 708	73 665	146 754	72	293	1 176
Carry Save	20 159	33 319	58 674	65	276	1 140
Booth radix 2	38 563	71 084	148 704	85	309	1 873
Booth radix 4	24 558	42 359	77 961	61	228	879
Booth radix 8	26 227	38 825	61 893	101	479	1 889

Příloha 15 - parametry násobiček, ASIC implementace, optimalizace syntézy: rychlost

Algoritmus	Výpočetní zpoždění [ps]			Plocha struktury [ekviv. hradel]		
	8 bitů	16 bitů	32 bitů	8 bitů	16 bitů	32 bitů
AGM	5 769	13 896	26 820	327	1 265	4 603
Shift And Add	8 808	19 325	40 203	359	1 435	5 941
Ripple-Carry	8 862	19 300	40 357	333	1 435	5 941
Carry Save	6 366	13 453	27 628	333	1 435	5 941
Booth radix 2	11 476	26 493	55 194	572	2 289	8 994
Booth radix 4	9 437	22 443	46 899	678	270	10 833
Booth radix 8	9 905	27 052	56 245	909	4 494	18 236

Příloha 16 - parametry násobiček, ASIC implementace, optimalizace syntézy: plocha

Algoritmus	Výpočetní zpoždění [ps]			Plocha struktury [ekviv. hradel]		
	8 bitů	16 bitů	32 bitů	8 bitů	16 bitů	32 bitů
AGM	6 938	16 576	32 556	327	1 279	4 686
Shift And Add	10 096	22 098	46 102	333	1 435	5 941
Ripple-Carry	8 886	19 325	40 203	333	1 435	5 941
Carry Save	7 170	15 374	31 783	333	1 435	5 941
Booth radix 2	15 349	33 394	69 940	829	3 401	13 661
Booth radix 4	12 223	27 742	58 810	827	3 319	13 169
Booth radix 8	12 699	31 137	67 871	1 150	5 088	20 361

Příloha 17 - spotřeba násobiček, ASIC implementace, optimalizace syntézy: rychlost

Algoritmus	Klidová spotřeba [nW]			Dynamická spotřeba [mW]		
	8 bitů	16 bitů	32 bitů	8 bitů	16 bitů	32 bitů
AGM	74,54	353,19	1 322,99	3,18	25,96	126,63
Shift And Add	81,66	319,33	1 342,55	2,98	21,69	148,22
Ripple-Carry	71,68	319,33	1 342,55	2,90	21,58	147,46
Carry Save	71,68	319,33	1 342,55	3,01	21,06	136,88
Booth radix 2	149,05	607,87	2 467,47	5,89	39,16	261,01
Booth radix 4	173,05	764,20	3 042,55	6,62	52,36	380,32
Booth radix 8	231,54	1 239,96	5 061,53	8,78	92,01	692,54

Příloha 18 - spotřeba násobiček, ASIC implementace, optimalizace syntézy: plocha

Algoritmus	Klidová spotřeba [nW]			Dynamická spotřeba [mW]		
	8 bitů	16 bitů	32 bitů	8 bitů	16 bitů	32 bitů
AGM	74,54	352,46	1 322,97	3,23	25,70	128,78
Shift And Add	71,68	319,33	1 342,55	3,00	22,57	154,72
Ripple-Carry	71,68	319,33	1 342,55	2,94	21,83	148,76
Carry Save	71,68	319,33	1 342,55	3,12	22,11	145,89
Booth radix 2	213,20	865,29	3 541,37	9,71	71,29	506,20
Booth radix 4	228,67	917,94	3 663,95	9,28	68,91	510,43
Booth radix 8	289,79	1 328,01	5 381,30	11,84	104,64	827,75

Příloha 19 - parametry děličeček, FPGA implementace, optimalizace syntézy: rychlost

Algoritmus	Výpočetní zpoždění [ps]			Plocha struktury [slices]		
	8 bitů	16 bitů	32 bitů	8 bitů	16 bitů	32 bitů
RTZ	42 595	92 570	212 870	73	283	1 115
NRTZ	37 240	75 487	181 777	49	161	578
SRT radix 2	39 130	78 829	189 097	49	162	581

Příloha 20 - parametry děličeček, FPGA implementace, optimalizace syntézy: plocha

Algoritmus	Výpočetní zpoždění [ps]			Plocha struktury [slices]		
	8 bitů	16 bitů	32 bitů	8 bitů	16 bitů	32 bitů
RTZ	42 595	92 570	212 870	73	283	1 115
NRTZ	37 240	75 487	181 777	49	161	578
SRT radix 2	39 130	78 829	189 097	49	162	581

Příloha 21 - parametry děličeček, ASIC implementace, optimalizace syntézy: rychlost

Algoritmus	Výpočetní zpoždění [ps]			Plocha struktury [ekviv. hradel]		
	8 bitů	16 bitů	32 bitů	8 bitů	16 bitů	32 bitů
RTZ	29 354	117 675	464 115	461	1 835	7 424
NRTZ	31 825	127 842	448 782	459	1 790	7 000
SRT radix 2	31 065	130 380	479 510	516	2 217	8 532

Příloha 22 - parametry děličeček, ASIC implementace, optimalizace syntézy: plocha

Algoritmus	Výpočetní zpoždění [ps]			Plocha struktury [ekviv. hradel]		
	8 bitů	16 bitů	32 bitů	8 bitů	16 bitů	32 bitů
RTZ	37 311	144 759	551 485	485	1 995	8 107
NRTZ	21 976	79 604	291 712	739	3 099	12 688
SRT radix 2	40 240	146 800	534 178	900	3 596	14 407

Příloha 23 - spotřeba děliček, ASIC implementace, optimalizace syntézy: rychlost

Algoritmus	Klidová spotřeba [nW]			Dynamická spotřeba [mW]		
	8 bitů	16 bitů	32 bitů	8 bitů	16 bitů	32 bitů
RTZ	95,80	395,59	1614,25	7,61	60,14	421,59
NRTZ	120,46	537,50	2156,86	11,16	87,96	585,29
SRT radix 2	119,98	579,53	2299,44	9,60	79,73	542,96

Příloha 24 - spotřeba děliček, ASIC implementace, optimalizace syntézy: plocha

Algoritmus	Klidová spotřeba [nW]			Dynamická spotřeba [mW]		
	8 bitů	16 bitů	32 bitů	8 bitů	16 bitů	32 bitů
RTZ	97,93	403,02	1 635,03	8,37	71,13	499,26
NRTZ	170,27	714,01	2 921,12	12,64	111,69	794,56
SRT radix 2	203,95	825,47	3 307,99	15,43	135,55	882,74